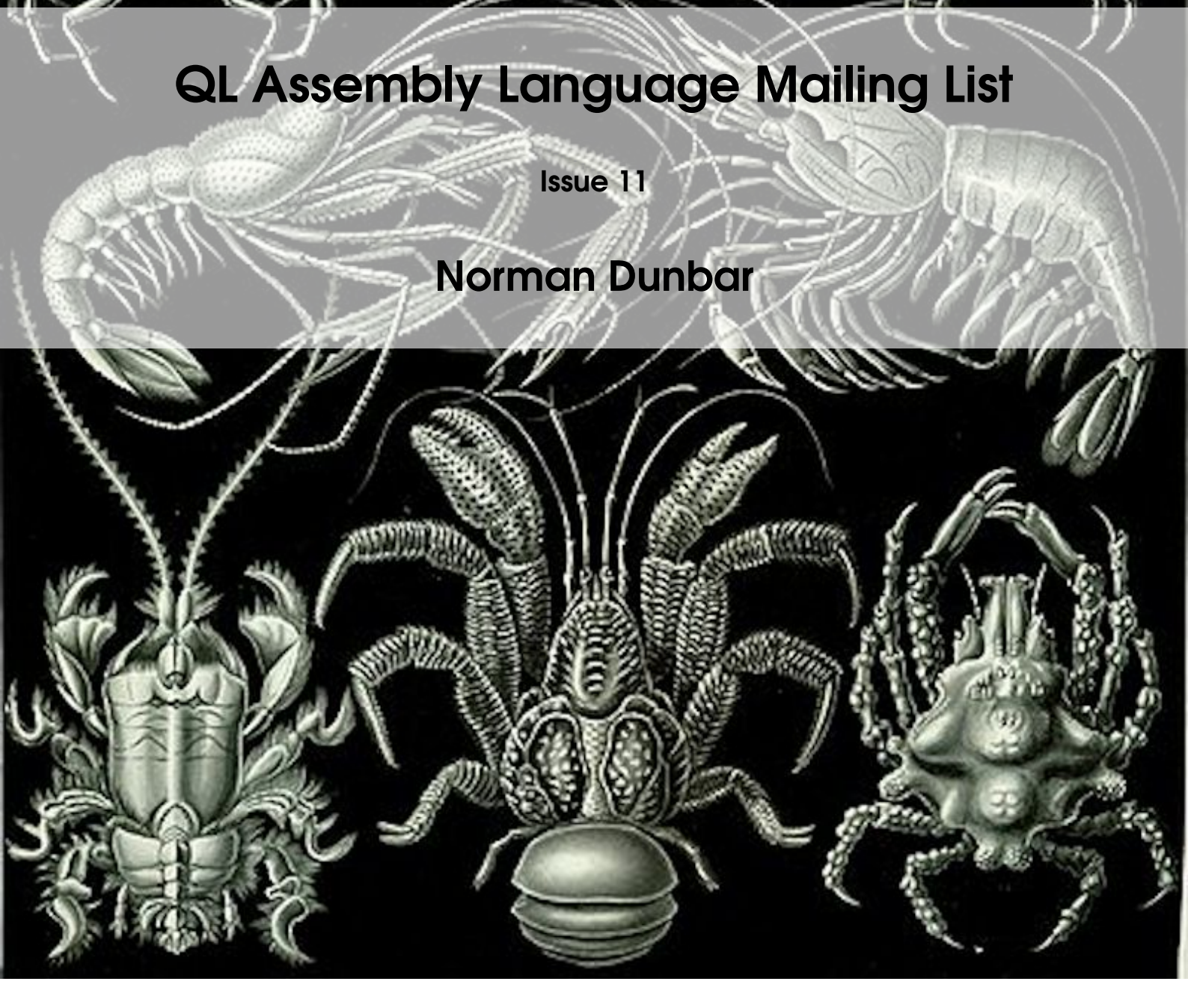


QL Assembly Language Mailing List

Issue 11

Norman Dunbar



PUBLISHED BY MEMYSELF EYE PUBLISHING ;-)

Download from:

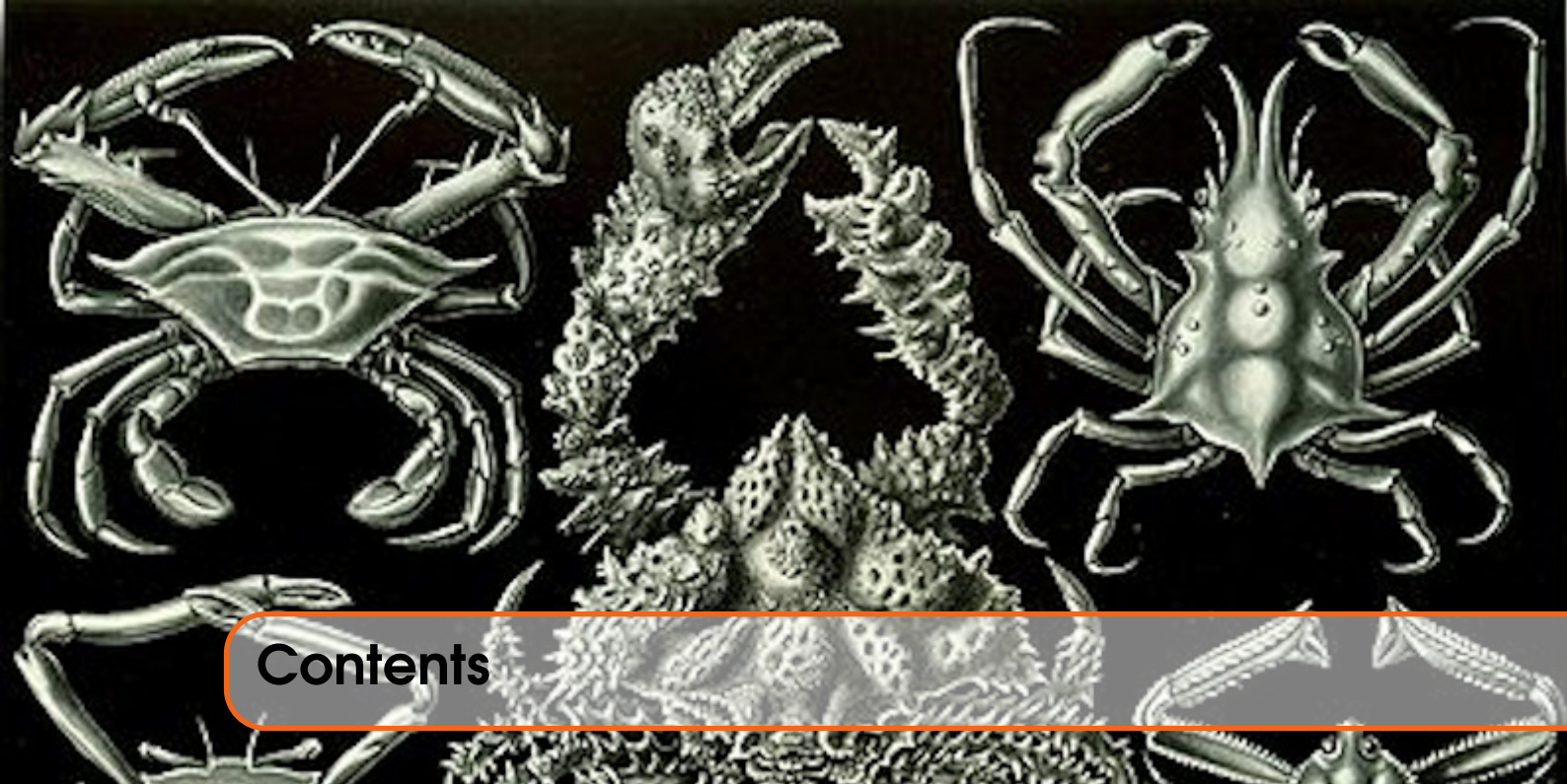
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_11

Licence:

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

This pdf document was created on *12/4/2022* at *14:26:09*.

Copyright ©2022 Norman Dunbar

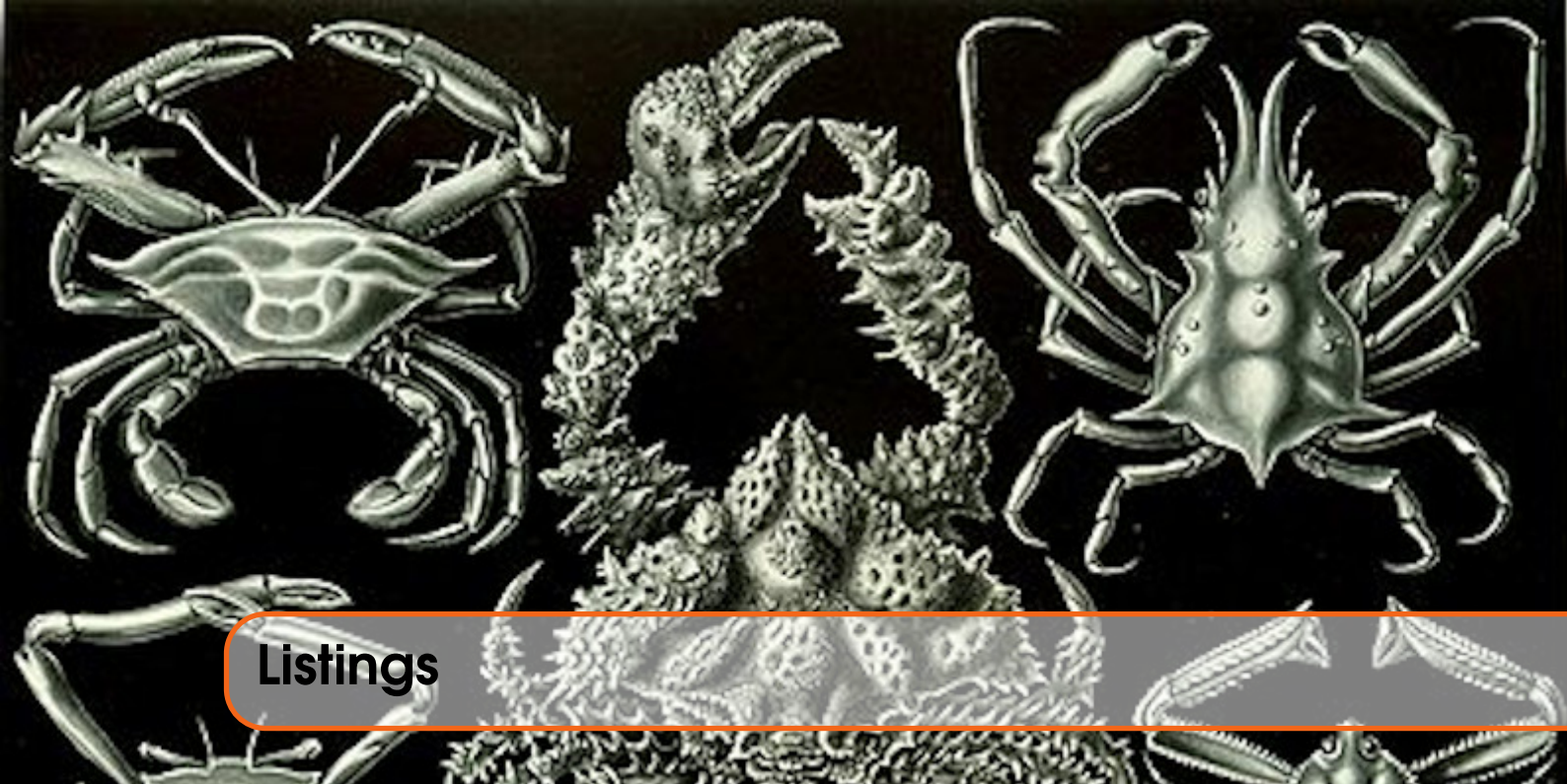


Contents

1	Preface	11
1.1	Feedback	11
1.2	Subscribing to The Mailing List	11
1.3	Contacting The Mailing List	12
2	News	13
2.1	Beginners Corner	13
2.2	Sudoku Solver	13
2.3	Multiplication	13
2.4	The QL Wiki	14
2.5	RIP GWASL	14
3	Beginners Corner	15
3.1	Introduction	15
3.2	Reading Files	15
3.2.1	Converting to Assembly	15
3.3	Code Walk-through	16
3.3.1	Preliminaries	16

3.3.2	Opening the Screen Channel	17
3.3.3	Clearing the Screen	18
3.3.4	Opening the Text File	19
3.3.5	Reading a Line of Text	20
3.3.6	Printing the Text	21
3.3.7	Suspending the Job	22
3.3.8	Closing Files	23
3.3.9	Done!	23
3.3.10	Subroutine(s)	24
3.4	Assembling the Code	24
3.4.1	With GWASS	24
3.4.2	With Qmac	24
3.4.3	Executing the Code	25
3.5	Assembler Differences	25
3.6	Whither GWASL?	26
3.7	Summary	27
4	Quickie Corner	29
4.1	It's All About Nothing!	29
5	Sudoku Solver	31
5.1	Quick Explanation	32
5.2	The Solver	32
5.3	The Solver Code	33
5.3.1	Equates	33
5.3.2	Job Header	34
5.3.3	Main Control Code	34
5.3.4	Printing the Board	35
5.3.5	Solving the Board	37
5.3.6	Is Digit in this Row?	39
5.3.7	Is Digit in this Column?	40
5.3.8	Is Digit in this Box?	41
5.3.9	Subroutines	42

6	32 Bit Multiplication	53
6.1	The MC68008 and MC68020	53
6.2	The Theory	53
6.3	The Code	54
6.4	What About Signed Values?	57
6.5	A Final Warning	58
6.6	And Finally	59
7	Image Credits	61



Listings

3.1	ReadingFiles - Constants	16
3.2	ReadingFiles - SMSQ/E Constants	16
3.3	ReadingFiles - Job header	17
3.4	ReadingFiles - Screen definition	17
3.5	ReadingFiles - Open screen channel	18
3.6	ReadingFiles - Clearing the screen channel	18
3.7	ReadingFiles - Opening the text file	19
3.8	ReadingFiles - Text file definition	19
3.9	ReadingFiles - Reading a buffer of text	20
3.10	ReadingFiles - Input buffer definition	20
3.11	ReadingFiles - Text file error handling	20
3.12	ReadingFiles - Buffer conversion to a string	21
3.13	ReadingFiles - Printing the data	22
3.14	ReadingFiles - Suspending the job	22
3.15	ReadingFiles - Closing all files	23
3.16	ReadingFiles - Death of a job	23
3.17	ReadingFiles - Closing one file	24
5.1	Equates	33
5.2	Job Header	34
5.3	Main Control Code	34

5.4	Printing the Puzzle Grid	35
5.5	Solving the Puzzle Pseudo Code	37
5.6	Solving the Puzzle Grid	38
5.7	Checking if a digit is in a row	39
5.8	Checking if a digit is in a column	40
5.9	Checking if a digit is in a box	41
5.10	Subroutine: Print a byte	42
5.11	Subroutine: Position the cursor at 2,0	43
5.12	Subroutine: Printing strings	43
5.13	Subroutine: Console definition and open code	43
5.14	Subroutine: Menu	44
5.15	Subroutine: Clearing bits of the screen	45
5.16	Subroutine: Loading a puzzle - prompting the user	46
5.17	Subroutine: Loading a puzzle - opening the file	46
5.18	Subroutine: Loading a puzzle - setting up the read	46
5.19	Subroutine: Loading a puzzle - reading data	47
5.20	Subroutine: Loading a puzzle - closing the data file	47
5.21	Subroutine: Getting user input	48
5.22	Subroutine: Scanning for ESC	48
5.23	Subroutine: KEYROW command for ESC	49
5.24	Subroutine: Demo puzzle data	49
5.25	Subroutine: Grid horizontal separator	50
5.26	Subroutine: Messages	50
6.1	Unsigned multiplication - initialisation	55
6.2	Unsigned multiplication - testing for zero	55
6.3	Unsigned multiplication - step 1	55
6.4	Unsigned multiplication - step 4	55
6.5	Unsigned multiplication - step 2	56
6.6	Unsigned multiplication - step 2 addition	56
6.7	Unsigned multiplication - step 3	56
6.8	Unsigned multiplication - step 3 addition	56
6.9	Unsigned multiplication - restoring D0	57
6.10	Unsigned multiplication - all done	57
6.11	Signed multiplication - initialisation	57

6.12 Signed multiplication - testing D0	57
6.13 Signed multiplication - testing D1	57
6.14 Signed multiplication - doing the multiplication	58
6.15 Signed multiplication - adjusting the result	58
6.16 Signed multiplication - termination	58
6.17 MC68020 - unsigned 32bit multiplication	59
6.18 MC68020 - signed 32bit multiplication	59



1. Preface

1.1 Feedback

Please send all feedback to assembly@qdosmsq.dunbar-it.co.uk. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in \LaTeX source format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to <http://qdosmsq.dunbar-it.co.uk/maillinglist> and clicking on the link "Subscribe to our Newsletters".

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

assembly@qdosmsq.dunbar-it.co.uk

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a \LaTeX source document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.



2. News

2.1 Beginners Corner

In Beginners Corner this time, we take a look at simple file and screen handling. The code to open a text file and to display it on screen is provided with explanations in excruciating detail!

Buffer overruns are handled safely and without affecting the program's output. EOF is a little more tricky in Assembly than in S*BASIC, and this is – hopefully – well explained too.

2.2 Sudoku Solver

In this issue, the main article is all about solving Sudoku puzzles using the QL and Assembly Language. This demonstrates recursion and backtracking and uses a brute force “algorithm” to solve the puzzles.

Well, I say “solve” but in reality, it stuffs a number into the first blank cell it finds, and then tries, recursively, to solve the new puzzle which has one less blank cell! There's no finesse about it, it just hammers away until it – eventually – solves the puzzle, or doesn't if the puzzle can't be solved.

2.3 Multiplication

After many years of avoiding the matter, I've finally decided to look into how the QL could do multiplication of two 32 bit numbers to give a 64 bit result. It turns out to be a lot less mind numbing than I thought it would be. So, anyway, if I had to figure it out for myself, you might as well profit from my trials and tribulations!

In this issue, there's a whole chapter about multiplying two 32 bit *unsigned* values to give a 64 bit result. Then I figured out that I might as well add an extra bit of code to make signed number multiplication “easy” as well. That turned out to be only a few more lines.

2.4 The QL Wiki

For some time now, there has been a problem uploading and/or including graphics on the [QL Wiki](https://qlwiki.qlforum.co.uk)¹ in that sometimes it works, but on others, it barfs with an “Error 403, Forbidden” message. This affects me as well, and I have administrator privileges on the Wiki.

Investigations are ongoing, and sadly, have been for quite some time. We are of the opinion that the host company has done something and barfed things for the Wiki. Rob Heaton is *still* attempting to get some form of support out of the hosts, but so far, this is proving quite elusive, unfortunately.

This is not a good state of affairs to be in, but at the moment, we are working on it as best we can, given the limitations of the hosts. They did apparently resolve it a while back, but it has reared its ugly head once again, but this time, we seem to have stumped them.

If you are trying to insert images into your Wiki pages, and are suffering from this problem, please be patient. There is a [Wiki Topic](https://qlforum.co.uk/viewforum.php?f=26)² on the QL Forum where you can report problems with images. Just to keep prodding us admin types!

Occasionally I am able to get files uploaded, with a bit of hassle from the “403 Forbidden” errors, but it can, eventually be done. If you need to get some images uploaded, let me know on the above link and I’ll try my best to get them done for you.

2.5 RIP GWASL

My most used assembler these days is GWASS from George Gwilt which allows me to assemble MC68020 instructions as well as those for the MC68008. In the past I have used, and occasionally still do, George Gwilt’s other fine assembler, GWASL. This one is only for the MC68008 processor used in the original QL.

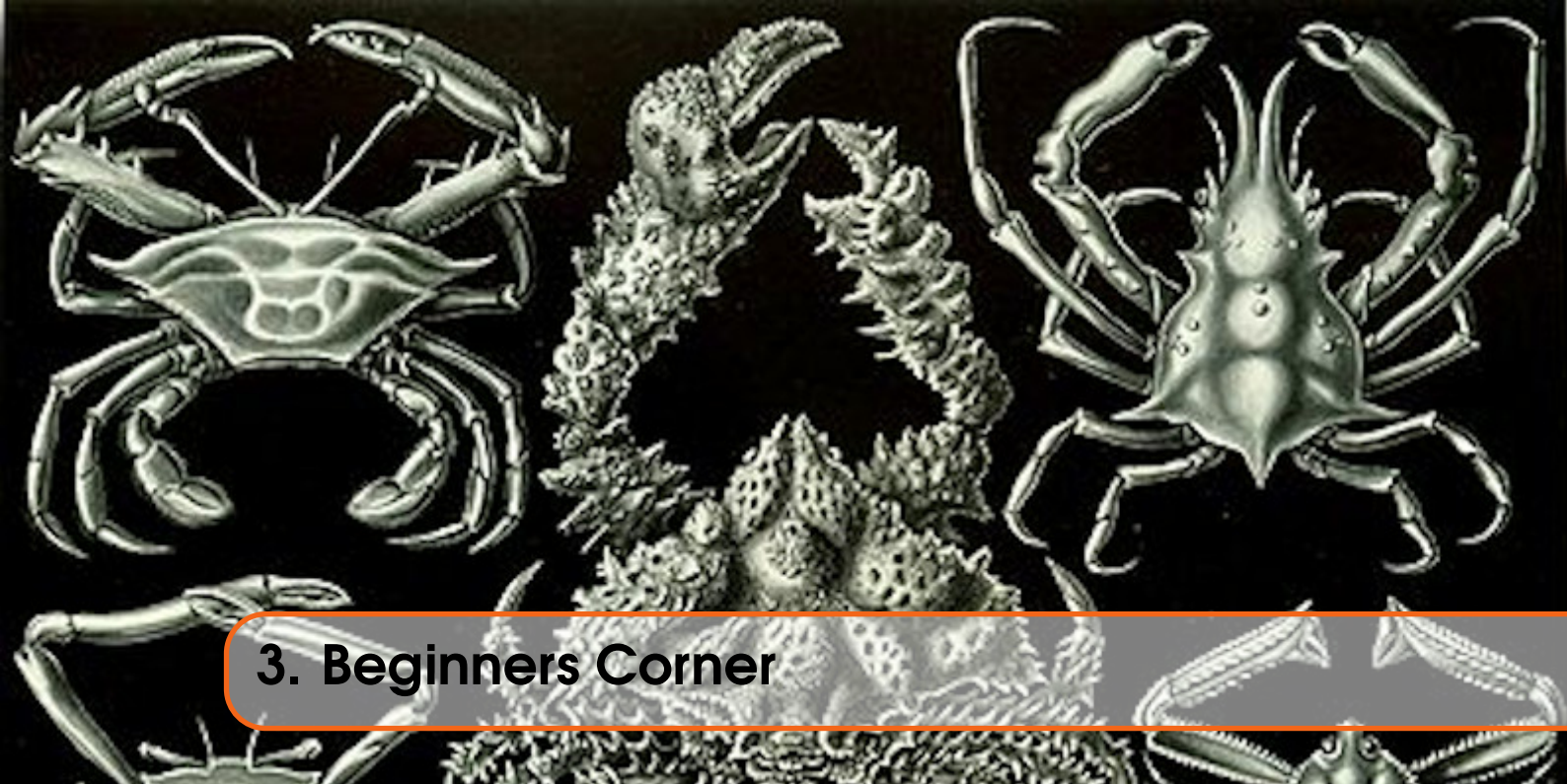
Unlike its big brother, GWASS, GWASL throws errors if a “blank” line in the source code is not completely blank. If there are any unprintable characters – spaces or tabs etc – GWASL will record an “illegal instruction” error when it hits one of these lines.

In addition, GWASL doesn’t like symbols with a dot in the name. So `OPW.CON`, for example, is rejected. This means that it’s not possible to use the correct names for the SMSQ/E symbols and equates.

Because of this, GWASL, which I have used since 2009, is sadly being retired. I do have the source code for GWASL but looking through it, George has used some of his own libraries and macros for various parts and I don’t have those files. I *might* do a disassembly on the current binary, to see if I can fix it. No promises though!

¹<https://qlwiki.qlforum.co.uk>

²<https://qlforum.co.uk/viewforum.php?f=26>



3. Beginners Corner

3.1 Introduction

In this issue of Beginners Corner, we are taking a look at file and simple screen handling.

3.2 Reading Files

This code in S*BASIC looks simple enough, and *should* be able to be converted quite simply to assembly, so let's do it.

```
OPEN #3, "ram1_test_txt"  
REPEAT readLoop  
  IF EOF(#3) THEN EXIT readLoop: END IF  
  a$ = INPUT(#3)  
  PRINT a$  
END REPEAT readLoop  
CLOSE #3
```

3.2.1 Converting to Assembly

The following steps are required in the assembly language source code:

- Decide whether the code will be CALLED or EXEC'd.
- Set up a buffer to hold the data we read from the file. How big should we make it? How long is a piece of string¹?
- Open a channel to the screen. This can be a CON_ or a SCR_ channel. SCR_ is most appropriate as we don't need to fetch any input from it.

¹Silly question! It's obviously twice as long as half a piece of string!

- We can set the paper, strip and ink colours as required. The default green on black isn't very nice on my old eyes, so that's got to go!
- Clear the screen channel.
- Open the file and handle errors if it's not found.
- Loop around, reading text from the string and printing it, exiting when end of file is detected.
- Close the file at the end of the loop.
- Exit back to S*BASIC.

As I prefer setting up and using separate jobs, I'll make this a job and it will be EXEC'd.

3.3 Code Walk-through

As this is Beginner's Corner, the following code fragments will be explained on excruciating detail. Experienced coders should probably look away now!

The full code is in the code repository for this issue.

3.3.1 Preliminaries

The code file, `ReadingFiles_asm`, begins with some constants – *equates* – which make life easier if we have to make changes at some future point. They also have reasonably explanatory names, which helps document the code. Listing 3.1 is the constants used in defining the screen channel we wish to use, and a couple more to define buffer sizes, to indicate the current job identifier and an infinite timeout.

```

TIMEOUT      equ  -1          ; Infinite timeout
ME           equ  -1          ; Current job id

WHITE        equ  7          ; White colour
BLACK        equ  0          ; Black colour

BORDCOLOUR   equ  WHITE      ; Border colour
BORDWIDTH    equ  1          ; Border width
PAPER        equ  BLACK      ; Paper colour
INK          equ  WHITE      ; Text colour

CON_W        equ  310         ; Console width
CON_H        equ  120         ;      "      height
CON_X        equ  50          ;      "      X position
CON_Y        equ  50          ;      "      Y position

BUFFERSIZE   equ  30         ; Input buffer size (in words)

OPEN_IN      equ  1          ; File mode, OPEN_IN

```

Listing 3.1: ReadingFiles - Constants

Following on from the program constants, we have a list of the various trap and vector codes to enable us to access routines within SMSQ/E and QDOS. Even though I'm using the SMSQ/E names, that code will still work on QDOS as they are the same as QDOS. Listing 3.2 shows the list of SMSQ/E constants we need in this program.

```

OPW.SCR      equ  $c8         ; Open a screen , border , etc
IOW.CLRA     equ  $20         ; CLS

```



```
IOA.OPEN    equ $01          ; Open a file
IOA.CLOS    equ $02          ; Close a file
ERR.EOF     equ -10         ; End of File error
ERR.BFFL    equ -5          ; Buffer overflow error
ERR.OVFL    equ -18         ; Overflow error
SMS.SSJB    equ $08         ; Suspend a job
SMS.FRJB    equ $05         ; Force remove a job
IOB.FLIN    equ $02         ; Fetch a line of text plus LF
UT.WTEXT    equ $d0         ; Print some text
```

Listing 3.2: ReadingFiles - SMSQ/E Constants

And now the code proper begins. As I have decided to create a separate job (or task) for this demonstration, we need a standard job header. Listing 3.3 is the necessary code for just about any job in SMSQ/E.

```
Start
  bra.s openScreen          ; Skip over job header
  dc.l 0                    ; 4 bytes, can be any value
  dc.w $4afb                ; Job flag, must be $4afb

jobName
  dc.w jobNameEnd-* -2      ; Length of job name
  dc.b "Reading Files "     ; Bytes of job name
jobNameEnd
  equ *

  ds.w 0                    ; Ensure address alignment
```

Listing 3.3: ReadingFiles - Job header

At label `Start`, the code skips the header and branches off to the `openScreen` label. Following that we have a 4 byte filler followed by the word `$4AFB` which SMSQ/E uses to recognise a job header. Following the flag word, we have the job's name in the form of a standard SMSQ/E string with a leading byte counter followed by the bytes of the string.

The strings length word might look a bit strange, however, we simply take the address of label `jobNameEnd`, subtract the current address (that of `jobName`) from it, then subtract another 2. This gives me the length of the text making up the job name. It also means that I can change the job name at any time and not have to worry about counting characters!

Tip: The `'*`' used in this way, tells the assembler to use the current address.

The last line in Listing 3.3 simply makes sure that the next address will be even if the job name contained an odd number of characters. It reserves zero words of space, but because this is word sized, the assembler forces the current address to an even address if necessary.

That completes the preliminaries of the job. Next we move on to the code proper, opening the screen channel.

3.3.2 Opening the Screen Channel

Listing 3.4 is the definition of the screen channel we wish to open. It consists of 4 bytes defining colours and border sizes, followed by 4 words defining the screen channel size. The values used here were defined as constants in Listing 3.1 and can be changed there if necessary.

```
scrDefine
```

```

dc .b BORDCOLOUR      ; Border colour
dc .b BORDWIDTH       ; And width
dc .b PAPER           ; Paper/strip colour
dc .b INK             ; Ink colour
dc .w CON_W           ; Width
dc .w CON_H           ; Height
dc .w CON_X           ; X pos
dc .w CON_Y           ; Y pos

```

Listing 3.4: ReadingFiles - Screen definition

The definition block is used by the `OPW.SCR` vectored routine to open a channel to the screen, set the paper, strip and ink colours, as well as setting the border colour and width. Listing 3.5 is the code to do exactly that.

```

openScreen
  lea scrDefine , a1      ; Parameters
  move.w OPW.SCR, a2     ; Scr_ channel required
  jsr (a2)               ; Open Console & set colours
  bne die                ; If it failed – bale out
  move.l a0 , a5         ; Save channel id for screen

```

Listing 3.5: ReadingFiles - Open screen channel

The `OPW.SCR` vector requires the address of the definition block in register A1 and when executed will return the channel id in register A0 – most channel functions use A0 for the channel id – and there will be an error code, or zero, in D0. If there were no errors, the Z flag will be set and we test this to ensure we can continue.

In the event of errors, we exit the program via the `die` routine, described later. Otherwise, we copy the screen channel id into register A5 as the code coming later to print to the screen will require the screen channel id in A0 and at that time, A0 will be holding the text file channel id.

3.3.3 Clearing the Screen

After opening the channel, we should clear the screen to get rid of any existing detritus that might be in the space where we opened the channel. Listing 3.6 shows the code to clear the entire screen channel. SMSQ/E allows different parts of the screen to be cleared, but we need to clear all of it.

```

clsScreen
  moveq #IOW.CLRA, d0    ; CLS whole window
  moveq #timeout , d3    ; Infinite timeout
  move.l d3 , d6         ; Save timeout for later
  trap #3                ; Do it
  tst.l d0               ; Any errors , probably not
  bne die                ; Yes, bale out

```

Listing 3.6: ReadingFiles - Clearing the screen channel

This code requires the screen channel in register A0, where it currently is, and a timeout in register D3. We use an infinite timeout in this case as we wish the action to complete before returning to our code. Most SMSQ/E trap calls preserve the timeout in D3, but when we open the text file, and when we print data to the screen, D3 will be corrupted, so we take a copy into register D6 for safety.

In the unlikely event of errors being detected when clearing the screen, we exit the job via the routine at label, `die`.

3.3.4 Opening the Text File

Now the good stuff! We get to open the text file named `ram1_test.txt`, which must exist. In order to do this, we need to have certain registers set as follows:

- The trap code, `IOA.OPEN` should be in `D0`;
- A job id, for the job to own the channel, is in `D1`;
- The open mode is in `D3`;
- `A0` points at the standard SMSQ/E string defining the job name;

There are a number of open modes that we can use when opening a file:

- `0 = OPEN` - the file specified must exist;
- `1 = OPEN_IN` - the file specified must exist;
- `2 = OPEN_NEW` - the file specified should not exist;
- `3 = OPEN_OVER` - the file specified may, or may not exist;
- `4 = OPEN_DIR` - opens an existing directory.

We are using the `OPEN_IN` mode in this example, so the file must exist. This is handy as we are intending to read and display its contents!

Listing 3.7 shows the code that opens the file. In the event of errors, the job simply exits in the usual manner, via `die`.

```
openFile
    moveq #IOA.OPEN, d0      ; Trap code, open a file
    moveq #ME, d1           ; I own the channel
    moveq #OPEN_IN, d3      ; OPEN_IN mode
    lea fileDefine, a0      ; File name
    trap #2                 ; Open File
    tst.l d0                ; OK?
    bne die                 ; No, bale out
```

Listing 3.7: ReadingFiles - Opening the text file

On return from the trap, the channel ID will be in register `A0`. This has of course overwritten the screen channel id that we have, but as we kept a copy of that in `A5`, it's not a problem. Listing 3.8 shows the text file's definition in the code. This is slightly out of line, as the definition is just below the screen definition block in Listing 3.4 but is documented here for clarity.

```
fileDefine
    dc.w fdEnd--2          ; Filename length
    dc.b "ram1_test.txt"   ; Filename bytes
fdEnd
    equ *                  ; End of filename

    ds.w 0                 ; Preserve alignment
```

Listing 3.8: ReadingFiles - Text file definition

As with the job name, we calculate the size of the file name to ensure we get it correct, and so that we can easily change the file name, if necessary, without having to keep counting characters.

Once we have the file open, we can start a loop to read the contents.

3.3.5 Reading a Line of Text

Listing 3.9 is the code which reads a buffer full of text from the text file, into our program. How big is the buffer? How do we know?

In S*BASIC it's quite simple, just read a line of text into a string variable. S*BASIC will read as many characters as it can into the string, until it hits EOF or reads a linefeed. Our code has a fixed buffer size, and I've made it deliberately small in this demonstration, to show how a long line can be read without causing buffer overflows and perhaps, corrupting the job code in the process. Normally, a buffer would be big enough – whatever that means.

```
readFile
  move.l d6, d3          ; Restore the timeout
  moveq #IOB.FLIN, d0    ; Fetch input with Linefeed
  move.w #BUFFERSIZE, d2 ; How big is my buffer?
  lea inputBuffer+2, a1  ; Input buffer space
  trap #3                ; Fetch input D1.W = size
  tst.l d0               ; Errors?
  beq.s setString       ; Nope
```

Listing 3.9: ReadingFiles - Reading a buffer of text

As mentioned above, opening the text file corrupted D3, so we copy it back from the safe place, register D6. The registers required to read a line of text, terminated by a linefeed, from a file, are:

- The trap code, IOB.FLIN, should be in D0;
- D2 should hold the buffer size – the maximum number of characters we can read;
- D3 is the timeout;
- A0 is the channel id;
- A1 points at the location in the buffer where the text will be copied into.

As we intend to print the text we read from the file, we set A1 to be two bytes higher than the start of the buffer. This leaves space to store the string length word prior to printing. This converts the buffer into a standard SMSQ/E string and makes for easy printing.

On return from the trap, we test for errors and if none were found, we skip to label setString, where we store the data length in the two empty bytes at the beginning of the buffer, to make the data into a string, ready to print. On return from this trap, the following registers are set:

- D0 has the error code, or zero;
- D1.W has the size of the data successfully read, which includes the linefeed, if one is present;
- A1.L points to the first byte after the last byte of data read, within the buffer.

Listing 3.10 is again, out of line with the actual code file, it comes at the very end, but it is the buffer definition and is relevant to this discussion.

```
inputBuffer
  ds.w    BUFFERSIZE+2      ; Buffer for data input
```

Listing 3.10: ReadingFiles - Input buffer definition

The size of the buffer is defined in our constants, way back in Listing 3.1. I've deliberately kept it small for this demonstration, as I previously mentioned.

In the event that we hit an error when reading the text file, what do we do? Listing 3.11 shows the error handling code.

```
cmpi.l #ERR.BFFL, d0      ; Overflow?
```

```

beq.s  setString          ; Yes, print what we got
cmpi.l #ERR.OVFL, d0     ; Other overflow?
beq.s  setString          ; Yes, print what we got
cmpi.l #ERR.EOF, d0      ; End of file?
tst.w  d1                 ; Did we read anything?
beq.s  suspendJob        ; No, close the file

```

Listing 3.11: ReadingFiles - Text file error handling

If D0 had been zero, then we would have successfully read a line of text, terminated by a linefeed, which was smaller (or equal) to the maximum buffer size allowed.

If, on the other hand, no linefeed was found before we filled the buffer, then we should receive ERR.BFFL on return from the trap. This indicates that we filled the buffer. This is not a proper error, it just means we have an extra long line in the file which is bigger than the buffer. We can still print it, so we can skip to setString.

The same is true if we detected ERR.OVFL. This is something that Wolfgang Lerner advised me to test for, in the last issue, as sometimes SMSQ/E (or QDOS?) can return this error instead of ERR.BFFL. The action is the same, it's not a proper error, we just filled the buffer, so we can skip to setString ready to print the data.

The next error we need to test for is end of file, ERR.EOF. If the end of file has been reached, did we read any data into the buffer? By testing D1.W we can tell. If there was no data read, D1.W will be zero and the Z flag will be set by the test. In this case, we skip to suspendJob as there's nothing else to do.

In assembly code, hitting EOF will raise the error *even if* some data were read into the buffer. In S*BASIC, the data would be returned with no errors and the following read of the file would raise the EOF error.

If there was data in the buffer, we still need to process it as normal, so we drop into Listing 3.12 where we set up the string, ready to be printed. This is described next. After printing, however, we will skip back to the start of the read loop, readFile, but this time, we will get the ERR.EOF error again but D1.W will be zero, so we have a get out of the loop clause!

3.3.6 Printing the Text

In order to print the string, we need the buffer to be in the style of a standard SMSQ/E string. This requires the string length to be in a word at the start of the string. When we read the data from the file, it was written to the third byte in the buffer, leaving room at the start for a word to hold the string (data) length.

On return from the IOB.FLIN trap, register D1.W has the number of bytes read. Listing 3.12 shows that word being stored at the start of the buffer, converting the buffer into a string, ready to print.

```

setString
  lea  inputBuffer, a1      ; The buffer start this time
  move.w d1, (a1)          ; Store the input size (inc LF)

```

Listing 3.12: ReadingFiles - Buffer conversion to a string

In the event that a linefeed was detected, printing the data will also print the linefeed, starting a new line in the screen channel for the next line of text. If no linefeed was found, the data will be printed, but no linefeed means that printing will begin again at the next space in the screen channel.

This way, long lines can be printed correctly as each buffer full of text is simply printed next to the previous one, until a linefeed is found.

Listing 3.13 prints the buffer as a standard string.

```
printInput
    exg a0 , a5                ; Swap file and screen ids
    move .w UT.WTEXT, a2      ; Print a string of bytes
    jsr (a2)                  ; Print it
    bne .s die                ; Ooops, error
    exg a0 , a5                ; Swap back again
    bra .s readFile          ; Loop around again
```

Listing 3.13: ReadingFiles - Printing the data

We currently have the channel id for the text file in A0 and we need the screen channel id for printing. As we kept a copy of that channel id in register A5, we can swap the two channel ids over. This means that A0 ends up with the screen channel id, and A5 the text file id.

We then load A2 with the vector for the UT.WTEXT routine, which expects the following parameters:

- A0 is the channel id;
- A1 is the start of the string to be printed, the length word.

After the code executes, D1-D3 and A1 are corrupted. D0 will hold zero or an error code. If errors are detected, we bale out of the job via the die routine. Otherwise, we swap the two channel ids back again, and skip back to the start of the reading loop to get the next line of text from the file.

3.3.7 Suspending the Job

This section of the code just delays closing the files for a few seconds so that the user can see the screen channel has the full text copied from the text file. It pauses for 300 frames – in the UK/Europe there are 50 frames per second, in the USA it's 60. This gives a 6 second delay in the UK/Europe, and 5 seconds in the USA. To suspend a job, you need the following register settings:

- The trap code, SMS.SSJB is in D0;
- The job id is in D1;
- The number of frames to suspend for is in D3;
- A1 is set to the address of a byte which will be cleared when the job's suspension is done and the job is active again. It can be zero if not required.

On exit from the suspension, register A0 is set to point at the job's header, while A1 is preserved.

Now, in my copy of Pennel, I have a hand written note saying *no it bloody isn't!* so I'm assuming I hit a problem some time back and noted it. Who is right? The documentation or "past" me? My motto is *Don't think, find out* so let's go! QMON2, here we come....

Setting a breakpoint at suspendJob, and running the job breaks at that label. If I set A1 to zero, then it is preserved after the suspension is over. So far, so good. What if A1 was not zero? Well, I tried that too – and it was also preserved. I'm running on SMSQ/E so maybe the problem I noted was in QDOS and has been fixed in SMSQ/E.

Anyway, Listing 3.14 shows the job suspension code.

```
suspendJob
    moveq #SMS.SSJB, d0        ; Suspend a job
    moveq #-1, d1              ; This job
    move .w #300, d3           ; 6 seconds is 300 frames
```

```

movea.l #0,a1          ; No byte to be cleared
trap #1                ; Suspend the job

```

Listing 3.14: ReadingFiles - Suspending the job

3.3.8 Closing Files

While it is not strictly required to close files when finished with them, it's considered good practice and should be done in applications where a file is opened and read once, perhaps at the start, but never used afterwards. In those jobs, holding the file open wastes resources, and can prevent the file from being edited to make changes.

The best advice is to keep files open only as long as you need to. However, I don't follow my own advice at times, so don't do as I do, do as I say!

Listing 3.15 shows the code to close both files. As we are doing the same thing more than once, we use a subroutine to close one file, and simply call it twice with the text file id, then with the screen channel id.

```

closeFiles
    bsr.s closeFile      ; Close the text file
    exg a0,a5           ; Get the screen channel id
    bsr.s closeFile      ; And close it too.

```

Listing 3.15: ReadingFiles - Closing all files

The first file to be closed is the text file, simply because we have the channel id in register A0, we call the subroutine `closeFile` to do the actual closing. Closing a file doesn't usually return errors. QDOS did return `ERR_NO` (SMSQ/E uses `ERR.ICHN`) for Channel not open, but SMSQ/E doesn't bother, and doesn't return an error.

On return, we swap A0 and A5 to get the screen channel id into A0, and head off to close that channel too.

The code to close one file is shown later, in Listing 3.17

3.3.9 Done!

Nearly done. The job has closed all its open files, and is ready to exit. We pass any error codes from D0 to D3, as is required, and call the `SMS.FRJB` trap call to kill the job and return the error code to S*BASIC. The parameters required are:

- The trap code, `SMS.FRJB` is in D0;
- The error code to return to S*BASIC is in D3.

Obviously, this trap call never returns.

```

die
    move.l d0,d3        ; Any errors?
    moveq #SMS.FRJB,d0  ; Force Remove a job
    moveq #-1,d1        ; -1 means "this job"
    trap #1            ; Kill this job

```

Listing 3.16: ReadingFiles - Death of a job

The error code will be reported by S*BASIC if, and only if, the job was started with `EXEC_W`, `EW` or any similar function which executes the job, then waits for it to complete.

Jobs started with EXEC, EX or similar, cannot report the jobs exit code as EXEC, EX etc return immediately after starting the job, they have no way of knowing how or if the job ended as some jobs can run “for ever” such as clocks. S*BASIC has no way of knowing how the job ended.

3.3.10 Subroutine(s)

There is a single subroutine in this issue’s code. The code to close a single file. Listing

3.17 shows the code. Closing a file requires these parameters:

- The trap code, IOA.CLOS in D0;
- The channel id in A0.

SMSQ/E returns no errors in D0 but QDOS does, if the file is not open, QDOS will return ERR.ICHN, which in QDOS terminology is ERR_NO.

```
closeFile
  moveq #IOA.CLOS, d0          ; Trap code, close file
  trap #2                     ; Close the text file first
  tst.l d0
  bne.s die
  rts
```

Listing 3.17: ReadingFiles - Closing one file

3.4 Assembling the Code

If I were to assume that you have downloaded [GWASS²](#) for QPC2 and other 68020 based emulators, or are using the Quanta version of the GST Qmac assembler for QLs with a 68008 processor, and have the code saved as ram1_ReadingFiles_asm, then assembling the code is as simple as this:

3.4.1 With GWASS

- EXEC gwass60_bin to start the assembler;
- Select the option to start assembling;
- Type in the filename: ram1_ReadingFiles_asm
- Wait.

3.4.2 With Qmac

To pass the commands directly via the command line:

- EX qmac;“ram1_readingFiles_asm -data 2048 -filetype 1 -nolink
-bin ram1_readingFiles_bin”

Note: The above command should be typed on one line - I’ve had to split it for the PDF page width.

Alternatively, you can type the command interactively:

- EX qmac
- Type the options: ram1_readingFiles_asm -data 2048 -filetype 1 -nolink
-bin ram1_readingFiles_bin

²<http://www.dilwyn.me.uk/asm/gwassp22.zip>

- Wait

What you are doing here, in both cases, is telling the assembler to:

- Assemble the source file `ram1_ReadingFiles_asm`;
- Create an executable file (`-filetype 1`), with 2,048 bytes of data space (`-data 2048`);
- Do not invoke the linker as it is not needed because everything is in the same source file (`-nolink`);
- Create the output file named `ram1_ReadingFiles_bin` – which will be in uppercase regardless of what you type here!

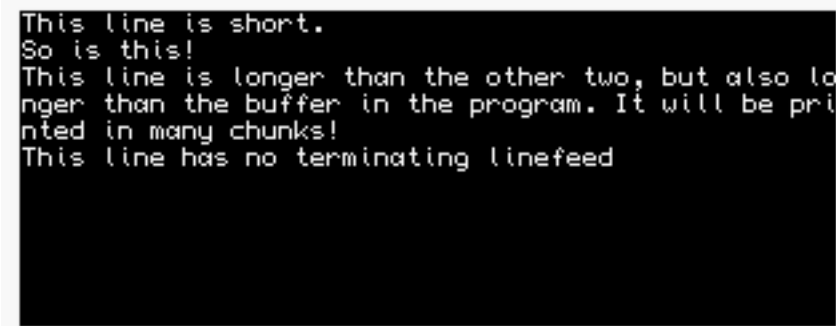
3.4.3 Executing the Code

After a successful assemble, and regardless of which assembler you used, `ram1_ReadingFiles_bin` will be the executable job. To run it:

- `EX ram1_ReadingFiles_bin`

On a successful execution, a small window will open, with black paper, white ink and a white, one pixel border. Some lines of text will be displayed, and after a delay of about 5 or 6 seconds, the screen channel will be closed and vanish.

And finally, Figure 3.1 shows what the final output should look like.



```

This line is short.
So is this!
This line is longer than the other two, but also longer than the buffer in the program. It will be printed in many chunks!
This line has no terminating linefeed

```

Figure 3.1: ReadingFiles final output

3.5 Assembler Differences

There are a couple of minor differences between assemblers, as you can see, just invoking them is quite different.

- Qmac doesn't have the ability to assemble instructions for the 68020 processor, only for the 68008 as used in the original QL. Some emulators use a virtual 68020 but these can still execute code assembled for the 68008, the converse is not true though.
- Qmac doesn't like filenames etc in double quotes, it expects single quotes. `"ram1_test_txt"` is rejected with the double quotes flagged as invalid characters, but `'ram1_test_txt'` is accepted.
- GWASS allows code without sections, but Qmac requires a header and trailer as follows:

```

section code
; Your code here

```

```
end
```

There needs to be a section with a name at the top, and an end at the bottom. You *can* have lots of sections, with different names but the above is the minimum necessary. GWASS doesn't need this.

- In Qmac (and GWASS), you can tell it how much data space it requires:

```
section code
data 8192

; Your code here

end
```

- Qmac has a command line option which can be used in make files, GWASS uses a non-standard method of passing filenames:

```
ex gwass60_bin; "A/1/filename_1_asm"
```

which tells GWASS to assemble 1 file, named filename_1_asm.

- Qmac doesn't like this sort of thing:

```
label
equ *
```

as it complains about the label being invalid. This is a bit of a bind as I use that a lot to show the end of a string and to save me counting. However, I can do this instead:

```
label equ *
```

which is accepted, and makes sure anything following is word aligned.

3.6 Whither GWASL?

This was mentioned already in the News section of this issue, but I'm repeating it here in Beginners Corner.

In the past I have used, and occasionally still do, George Gwilt's assembler, GWASL, for the MC68008 processor used in the original QL. Unfortunately, because GWASL throws errors whenever there is a blank line which contains one or more spaces or tabs or other non-printing characters; and cannot cope with the SMSQ/E symbols containing a dot in their name, GWASL is sadly now being retired.

Because of this, I've resorted to using QMAC for the MC68008 processors and GWASS for the bigger, MC68020, processors. QMAC has no problems with those "non-blank" blank lines or SMSQ/E names..

3.7 Summary

So that's the Assembly Language version of how to open a file, read it and display the contents on the screen, before closing it.

In future issues, I'll be continuing to delve into a few more of these with, hopefully, enough explanation for beginners to get started with.

Get hold of the SMSQ/E Reference Manual from [Wolfgang Lenerz's web site](#)³ for the official version. Alternatively, there are copies on Dilwyn's pages:

- [Here](#)⁴ for the PDF for version 4.5; or
- [Here](#)⁵ for the ODT (Libre Office) file for version 4.5.
- If you want to ensure that you have the most recent versions of those files, [Wolfgang Lenerz's web site](#)⁶ is the place to look.

Get hold of GWASS [here](#)⁷ for 68020 processors.

Download Qmac [here](#)⁸ for 68008 processors.

³<https://www.wlenerz.com/qlstuff/#qdosms>

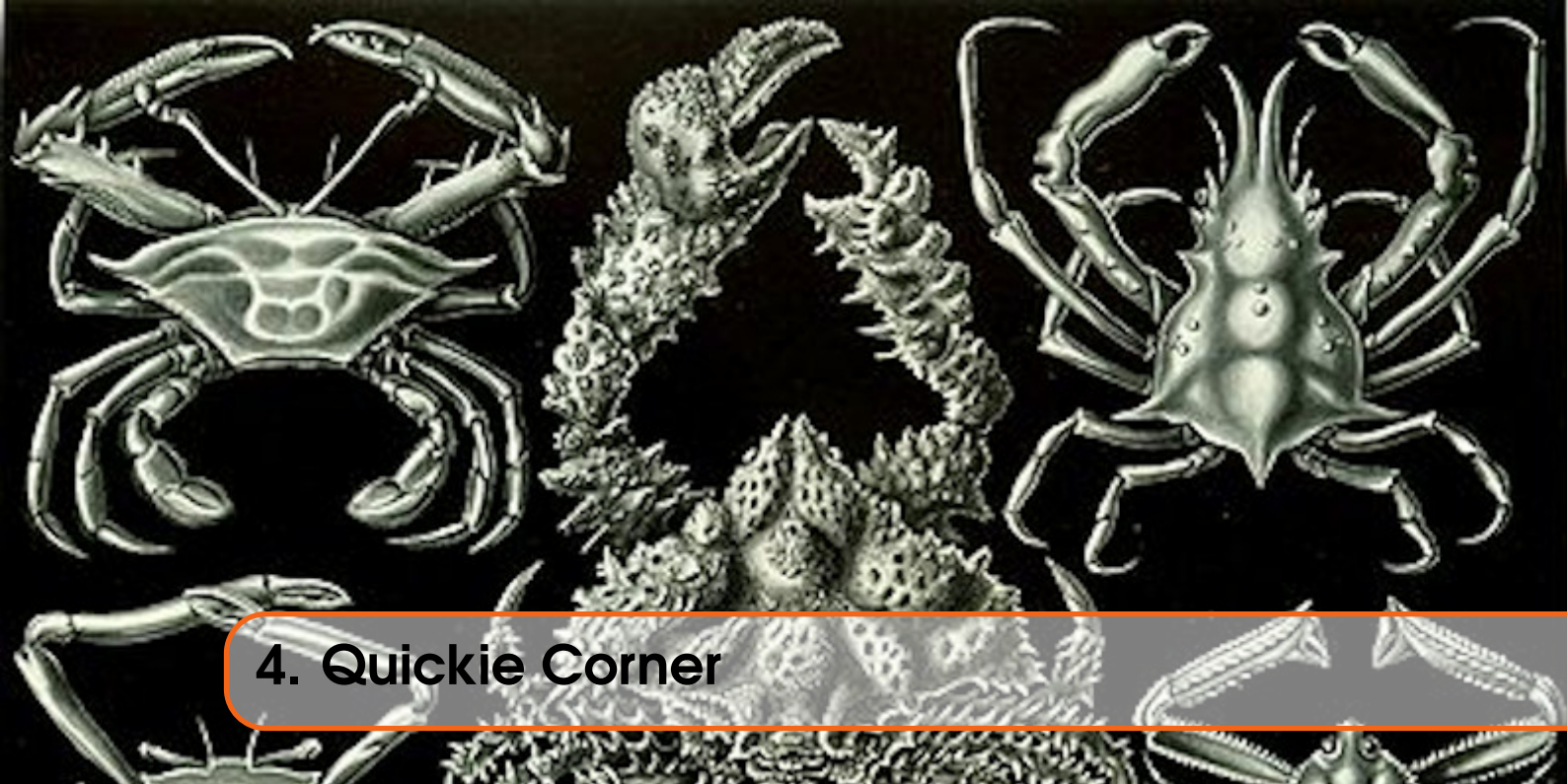
⁴http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.pdf

⁵http://www.dilwyn.me.uk/docs/manuals/QDOS_SMS%20Reference%20Guide%20v4.5.odt

⁶<https://www.wlenerz.com/qlstuff/#qdosms>

⁷<http://gwiltprogs.info/gwassp22.zip>

⁸<http://www.dilwyn.me.uk/asm/gst/gstmacroquanta.zip>



4. Quickie Corner

Following on from the resounding success¹ of last issue's Quickie Corner in which we found the fastest method of clearing a register to zero, this time we look at finding the quickest way to tell if a data register is zero, or not.

4.1 It's All About Nothing!

What's the fastest way to check if a data register is zero? The timings are taken from the Motorola Semiconductor's MC68000 Programmers Reference Guide, 4th Edition, Appendix E.

Table 4.1 shows a few instructions which can either test a data register for zero, or, cause the Z flag to be set if zero, and their timings.

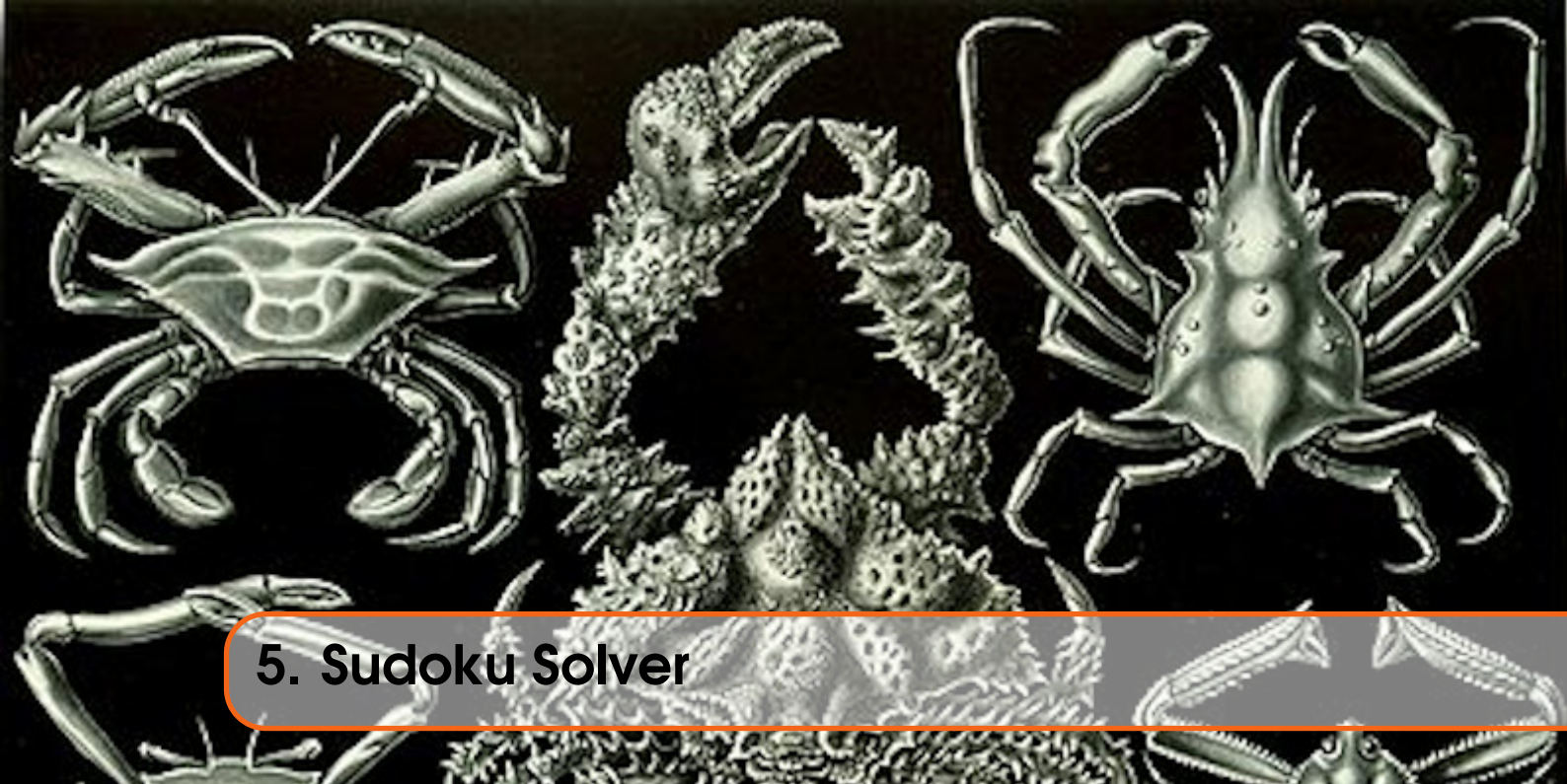
- Dn is the data register in question.
- Dz is a data register containing zero.
- Dy is "just another data register"

It appears that `TST.L Dn` is the quickest at 4 clock cycles, closely followed by `CMP.L Dn,Dz`, `NEG.L Dn`, `ADD.L Dz,Dn` and `SUB.L Dz,Dn` all at 6 clocks.

¹By which I mean, nobody complained!

Instruction	Clock Cycles
CMPL.L #0,Dn	12
CMP.L Dn,Dz	6
TST.L Dn	4
OR.L #0,Dn	14
OR.L Dn,Dn	22
ANDI.L Dn,Dn	14
AND.L Dn,Dn	22
MOVE.L Dn,Dy	4
NEG.L Dn	6
ADDQ.L #0,Dn	8
ADDI.L #0,Dn	14
ADD.L Dz,Dn	6
SUBI.L #0,Dn	14
SUB.L Dz,Dn	6
SUBQ.L #0,Dn	8

Table 4.1: Checking a data register for zero



5. Sudoku Solver

This is the most brutal program I think I've ever written. It's a Sudoku puzzle "solver" in as much as it will, eventually, find a solution if a puzzle can be solved, but it does it by brute force and complete ignorance of how a puzzle should be solved – using logic and deduction (induction?) – and, it's recursive!

To put it mildly, it starts with a grid from which some entries are missing, it looks for the one closest to the top left corner, and tries the digits 1 to 9 in that location. If the digit can be put there – it's not already in the same row, column or small box – then it effectively has a new grid to solve, and calls itself recursively, to solve the new board. If it successfully places another digit, then it recurses again and so on. Eventually though, the board will be solved, or a digit will not be able to be placed.

In the latter case, the recursive call will return – *backtrack* – to the calling code, reset it's variables, and try again with a different starting digit. This will progress until the puzzle is solved or every possible combination of digits for every possible blank space, has been tried.

Now, you might think this will need a lot of stack space for the recursive calls, but it needs $4 + (20 * depth)$ bytes of stack. By default, the solver has a 4,000 byte data space, and so far, I have yet to blow that out with any puzzles. That's enough for 199.8 recursive calls.

Did I mention, the code assembles to 1,028 bytes? And that includes a demo puzzle and the ability to load games from disc?

The idea for this "utility" came from a *You Tube* video¹ that was recommended to me, for some reason, by You Tube's so called AI, which seemed to think that I was interested in a Java Programming series. I'm not and I wasn't, but one of the offered videos was a Sudoku Solver in Java. So I stole the idea and wrote it in Assembly!

¹No, I don't have the URL I'm afraid, I didn't actually watch it – I just saw the description.

5.1 Quick Explanation

Picture a grid with three blank cells and the following explanation extremely simplified. The solution will progress as follows:

1. Try a '1' in the first cell,

1		
---	--	--

, if it is legal move, recurse to solve the grid with 2 blank cells;
2. A '1' in the new first cell will obviously be illegal, from the previous step, so try a '2',

1	2	
---	---	--

. If legal, recurse to solve the grid with a single blank cell;
3. Both '1' and '2' are illegal here, so all of the digits '3' through '9' will be tried until one is legal. If so, the grid is solved,

1	2	3
---	---	---

, and the code returns through all of the recursive calls to the top level, and finishes;
4. If none of the digits '3' through '9' are legal,

1	2	X
---	---	---

, then the grid is unsolved, so we return to the grid with two blank cells and try the remaining numbers – '3' through '9'. Starting with '3',

1	3	
---	---	--

, and another recursive call to see if the grid with one blank cell can now be solved;
5. If we find that we didn't get a solution to the two blank cells grid, we return back to the grid with three blanks, and try the next number that legally fits,

2		
---	--	--

, and recurse again to solve the new grid with two and then one blank cells;
6. This process repeats until we end up with either a solution, we fitted three legal digits in the three blank cells; or no solution – the grid is unsolvable.

5.2 The Solver

To use the solver, simply EX/EXEC it and you will be presented with a menu offering three choices:

- 'Q' or 'q' to quit;
- ENTER to run the built in demonstration puzzle;
- 'L' or 'l' to load a puzzle. You will be further prompted for the file name which should contain 81 bytes of grid data – the puzzle only caters for 9 by 9 grids at present – which can be a stream of 81 bytes or separated into 9 lines of 9 bytes using linefeeds. Digits are represented by themselves, obviously, and blanks are represented by whichever character you wish to use. Linefeeds are ignored but anything else that isn't a digit is deemed to be a space.
- After a puzzle is solved, or not, you are required to press ENTER to quit.

The load feature only accesses the first 81 bytes of data, ignoring linefeeds, to obtain the grid data. You can fill the file with anything you like after the first 81 bytes (or 90 if linefeeds are present).

The code repository for this issue contains a couple of example files to load – Sudoku_easy, Sudoku_hard and Sudoku_expert – but don't be misled, those are simply the puzzle levels from the Sudoku app on my tablet that I stole the grids from! In practice, the easy version takes longer for the brute force algorithm to solve than it does for the other two.

The code is written in plain 68008 instructions, there are no 68020 etc options used, so it will run on a bare bones QL. I have no idea how efficiently it will run though. My QPC setup – Linux under Wine, 64 bit dual core – takes a few seconds to solve the demo and the hard and expert loadable files, but longer of seconds to solve the easy loadable puzzle.

On with the code.

5.3 The Solver Code

The code is supplied in a format that makes it easy to assemble with my preferred assembler, GWASS, but with a couple of very minor changes, it can be assembled also by QMAC. The changes are just to remove two comment markers, a semicolon from the job header and the end of the file. All will be explained below.

5.3.1 Equates

At the start of the file, SudokuSolver_asm, we begin with a list of equates. Listing 5.1 shows them all in gory detail.

```

gridSize      equ 9          ; Grid is 9 by 9 usually
linefeed      equ 10         ; Linefeed character code
escKey        equ 3          ; Bit to test for ESC key
IOB.SBYT      equ 5          ; Print a byte trap code
IOW.SCUR      equ $10        ; Set cursor trap code
IOW.SINK      equ $29        ; Set ink colour
OPW.CON       equ $c6        ; Open CON channel
UT.WTEXT      equ $d0        ; Print string vector
SMS.FRJB      equ $05        ; Force remove a job
IOB.FLIN      equ $02        ; Get a line of text
IOW.CLRA      equ $20        ; CLS whole screen
IOW.CLRB      equ $22        ; CLS bottom of screen
IOW.CLRL      equ $23        ; CLS bottom of screen
IOA.OPEN      equ $01        ; Open a file
IOA.CLOS      equ $02        ; Close a file
IOB.FBYT      equ $01        ; Read one byte
SMS.HDOP      equ $11        ; Scan keyboard trap code

TIMEOUT       equ -1         ; Infinite timeout
ME            equ -1         ; Current job
BUFFERSIZE    equ 40         ; Input buffer length

WHITE         equ 7          ; White colour
GREEN         equ 4          ; Green colour
RED           equ 2          ; Red colour
BLACK        equ 0          ; Black colour

BORDCOLOUR    equ RED        ; Border colour
BORDWIDTH     equ 1          ; Border width
PAPER         equ GREEN       ; Paper colour
INK           equ BLACK       ; Text colour

CON_W         equ 160         ; Console width
CON_H         equ 180         ; " height
CON_X         equ 50          ; " X position
CON_Y         equ 50          ; " Y position

```

Listing 5.1: Equates

Most of these are fairly constant, but you might wish to adjust CON_X and/or CON_Y if you would like a different placement on the screen.

5.3.2 Job Header

The standard job header is next and Listing 5.2 shows it. The code is laid out in such a way that the “end” part of the job name is followed by an alignment byte, if necessary, to force the following address to be word aligned – even – if the job name happens to be an odd number of characters.

```

; section code ; Uncomment for QMC assembler
data 4000 ; Dataspace

Start
bra.s mainCode ; Skip over job header
dc.l 0 ; 4 bytes, can be any value
dc.w $4afb ; Job flag, must be $4afb

jobName
dc.w jobNameEnd--*--2 ; Length of job name
dc.b 'Sudoku Solver' ; Bytes of job name
jobNameEnd equ *

ds.w 0 ; Ensure address alignment

```

Listing 5.2: Job Header

The `equ *` is necessary on the same line as the label for QMAC to be able to assemble the code without errors and is also the reason why the job name is in single quotes – QMAC doesn’t appear to like double quotes² and throws errors.

You will notice a comment line, `section code`, near the start? This is needed to be uncommented for QMAC. Delete the semicolon at the start of the line if you are using the QMAC assembler.

5.3.3 Main Control Code

Following the job header, we have the main controlling code for the application. Listing 5.3 has the details.

We begin by calling `openScreen` to open the screen console we will be using, then printing the job name and the menu. On return from the menu, which will only occur if the user chose not to quit, we will start to solve whichever board we now have in memory. The code at `solveBoard` does this, and calls itself recursively as necessary in order to solve the puzzle.

```

mainCode
    bsr openScreen ; Open a CON_ channel
    lea jobName, a1 ; Pointer to jobname text
    bsr printString ; Print job name
    bsr menu ; Load, quit, demo menu
    lea board, a3 ; Start of the board
    bsr printBoard ; Print initial board
    bsr solveBoard ; Attempt to solve the board
    bne.s mcFailed ; Unsolvable board

mcSucceeded
    lea messSuccess, a1 ; Succeeded
    bra.s mcPrint

```

²Although a thought strikes me. I wonder if the double quotes I use in Linux, where I type the source code, is a different character to the QL double quotes? I must find out.

```

mcFailed
    lea messFailed , a1          ; Failed to solve board

mcPrint
    bsr printString             ; Print message

mcWaitForENTER
    lea messEnter , a1          ; Press ENTER message
    bsr printString             ; Print it
    bsr waitForInput            ; Wait for the ENTER key

;-----
; The job is complete , remove it from the system. No errors are
; returned to S*BASIC even with EXEC_W/EW.
;-----
die
    move.l #0, d3                ; No errors
    moveq #SMS.FRJB, d0          ; Force Remove a job
    moveq #me, d1                ; Kill this job
    trap #1                      ; Kill this job

```

Listing 5.3: Main Control Code

On return from the attempt to solve the board, we display a message regarding the success or failure of the attempt, then wait for the user to press ENTER and quit the program. The Z flag will be clear if the board cannot be solved, or if the user pressed ESC. Z will be set if the board was solved.

It would be a simple task to perhaps, make the code loop around and display the menu again, allowing the user to choose to load and solve another board? I leave this as homework and look forward to seeing your results.

5.3.4 Printing the Board

It's no good solving, or not, the puzzle but never displaying the results. Listing 5.4 is the `printBoard` code which does exactly that, it prints out the board converting any zeros in the board data into spaces and printing the vertical and horizontal separators to attempt to make the output resemble a Sudoku grid. All registers are preserved except D0 and on entry, the code expects register A3 to be holding the address of the start of the board data. The board is printed on every call to `solveBoard`.

The code should be easy to follow. It starts by setting the cursor position to row 2, column 0 on the screen, and using D6 as a loop counter for the 81 bytes of data to be processed. Each byte is loaded into D1 and if not a zero, must be a digit. The binary value in D1 is then converted to an ASCII digit by adding the code for the ASCII character '0' and is printed using the handy subroutine `printByte`.

If the value in D1 was a zero, we set D1 to the value of an ASCII space minus an ASCII zero. \$20 - \$30 in other words and drop in to convert back to an ASCII space, before printing it.

```

printBoard
    movem.l d1/d6/a3, -(a7)      ; Save important registers
    bsr at                       ; Cursor to 2,0
    moveq #0, d6                 ; Which byte next?

pbLoop
    move.b (a3)+, d1             ; Grab a byte

```

```

    bne .s pbNotZero          ; It's not a zero
    move .b #' '-'0',d1      ; Print a space instead

pbNotZero
    addi .b #'0',d1          ; ASCII digit
    bsr printByte            ; Print the byte in D1

pbTryLinefeed
    move .l d6,d1            ; Copy of current byte index
    divu #9,d1               ; Divide for linefeed
    swap d1                  ; Get remainder
    cmpi .w #8,d1            ; Remainder 8 = linefeed
    bne .s pbTryVseparator   ; No linefeed required

pbLinefeed
    move .b #linefeed ,d1
    bsr printByte            ; Print the linefeed

pbTryHSeparator
    cmpi .w #26,d6           ; Horizontal separator needed?
    beq .s pbHseparator      ; Yes
    cmpi .w #53,d6           ; Horizontal separator needed?
    bne .s pbLoopEnd         ; No

pbHSeparator
    lea hSeparator ,a1       ; Separator string
    bsr printString          ; Print it
    bra .s pbLoopEnd         ; No vertical separator needed

pbTryVSeparator
    move .l d6,d1            ; Another copy of index
    divu #3,d1               ; Divide for vertical separator
    swap d1                  ; Get remainder
    cmpi .w #2,d1            ; Remainder 2 = 'l' required
    bne .s pbLoopEnd         ; No vertical separator needed

pbVSeparator
    move .b #'l',d1          ; Vertical separator needed
    bsr printByte            ; Print it

pbLoopEnd
    addq .w #1,d6             ; Next index
    cmpi .w #81,d6           ; Are we done yet?
    bne .s pbLoop           ; No, keep going

pbDone
    movem .l (a7)+,d1/d6/a3   ; Restore board
    rts

```

Listing 5.4: Printing the Puzzle Grid

After printing a digit or space, we check to see if we need a linefeed after every 9th character printed. We copy D6, our cell counter, to D1 and divide by 9 and if the remainder is 8 we print a linefeed. We are counting from 0 to 80, so need a linefeed after the counter reached 8, 17, 26 and so on. After printing the linefeed we don't bother checking for vertical separators as they never occur at the end of a line, but we do need to consider the horizontal separators every 3 rows. These

occur when the cell counter is at value 26 or 53, and if so, we print the separator.

If a linefeed is not required, a vertical separator might be needed. The cell counter is again copied to D1 but this time divided by 3. If the remainder is 2, then we print a vertical separator otherwise, we don't print anything else.

At `pbLoopEnd`, we increment the cell counter and if not finished, skip back to the start of the loop to print the next cell's data. If we are finished, we restore the registers we needed to preserve, and return to wherever we were called from.

5.3.5 Solving the Board

Now we are ready to solve the board, and I use the term *solve* in its vaguest sense! The algorithm is as shown in listing 5.5, which is pseudo code for what I figured out would probably work!

```
Def FN solveBoard
  local r,c,tryMe
;
  for (r = 0; r < gridSize; r++) {
    for (c = 0; c < gridSize; c++) {
      if (board[r][c] = 0) {
        for (tryMe = 1; tryMe <= gridSize; tryMe++) {
          if (isValidHere()) {
            board[r][c] = tryMe;

            if (solveBoard(board)) {
              return true;
            } else {
              board[r][c] = 0;
            }
          } end if validHere
        } end for tryMe
        return false;
      } end if board[r][c] = 0
    } end for c
  } end for r
  return true
```

Listing 5.5: Solving the Puzzle Pseudo Code

We scan each row, and then scan each column within that row. If the board data holds a zero for this cell, then we have a blank cell for which we need a value. We iterate through each digit from 1 to 9 and check if it can be legally placed in this cell. If so, it is tried and a recursive call made to solve the new board with one cell less to find a value for.

If we have tried all the digits, and none fit, then the previous values attempted for one or more cells is not correct, so we return `false` to indicate failure, having set the cell data back to a zero.

The only way that we can return from this code having solved the board is if we have iterated through all rows and all columns and managed to legally place a digit there that caused all further recursive calls to also find a legal digit. The puzzle is solved, so we return a `true` value to indicate solved.

On return from a recursive call, if a `true` was detected, we also return `true` back to our caller, unwinding the stack as we go until we reach the top level.

How hard could it be to convert that into Assembly? Read on! Listing 5.6 shows the code I ended

up with, after much wailing and gnashing of teeth. I've added in a scan for the ESC key so that an attempt to solve a puzzle can be aborted if something has gone wrong and it's taking forever to find a solution.

If the scan returns with Z clear, then the ESC key was not pressed, so we can continue, otherwise we simply return to the caller with Z set to indicate a failure to solve the puzzle.

The code proper begins by saving our "local" variables, or registers in this case. If we are called recursively, which we will be, then we need to save the current values for row, column, and the digit we are trying to fit in. We also save the offset into the board data, register D4, as we need that to be restored before we return back from this recursive call.

D5 is the row counter, D6 is the column counter and D7 is the value we are attempting to insert into the current cell. D4, as mentioned, has the cell offset and is used as a way of not having to calculate the offset each time for each new row and column in the grid.

One problem, which I might look into, is the fact that every recursive call begins looking for blank cells from the *start* of the board. This is inefficient as surely, we should be looking one cell onwards from the cell we have just fitted a value into? More homework perhaps? Anyway, the offset, row and column registers are initialised and we loop along looking for a blank cell.

When we find one, D7 is initialised to 1 and we start looping around the values 1 to 9, checking each to see if it can legally be placed in the cell. A legal placement is when the value under test is not in the current row, as per D5, or column as per D6 or box, as per D5,D6. If the value in D7 is legal, it is placed in the puzzle data at offset D4, the board printed with the new value, and a recursive call made to `solveBoard` to try and solve this new grid.

If we return with Z set, the board has been solved and we return to our caller with Z set, otherwise, we reset the cell's data to zero and try another number at `sbNextNumber`.

```

solveBoard
    bsr scanForESC           ; Z clear = ESC pressed
    beq .s sbNoESC          ; Not pressed
    rts                     ; ESC

sbNoESC
    movem .l d4-d7, -(a7)    ; Save working registers
    moveq #0, d4            ; Offset into board data
    moveq #0, d5            ; Row number

sbRowLoop
    moveq #0, d6            ; Column number

sbColLoop
    tst .b (a3, d4.1)       ; Is this cell zero?
    bne .s sbNextColumn    ; No, skip this column
    moveq #1, d7            ; Otherwise, try 1 - 9 in cell

sbNumberLoop
    bsr .s inRow           ; Is D7 in row D5?
    beq .s sbNextNumber    ; Yes, skip
    bsr .s inColumn       ; Is D7 in column D6?
    beq .s sbNextNumber    ; Yes, skip
    bsr .s inBox          ; Is D7 in box D5/D6?
    beq .s sbNextNumber    ; Yes, skip

sbValid

```

```

    move.b d7,(a3,d4.1)      ; Put the number in the cell
    bsr printBoard          ; Update the board
    bsr.s solveBoard        ; Try to solve this new board
    beq.s sbSolved          ; Board is solved, hooray!
    clr.b (a3,d4.1)         ; Else, reset cell to zero

sbNextNumber
    addq.b #1,d7             ; Try next number
    cmpi.b #gridSize+1,d7   ; Unless we are done
    bne.s sbNumberLoop     ; Still going, loop again

sbUnsolved
    moveq #1,D0              ; No numbers fit here
    bra.s sbReturn          ; Bale out, failure

sbNextColumn
    addq.b #1,d4             ; Next cell offset
    addq.b #1,d6             ; Next column number
    cmpi.b #gridSize,d6     ; Done yet?
    bne.s sbColLoop        ; Loop over columns

sbNextRow
    addq.b #1,d5             ; Next row number
    cmpi.b #gridSize,d5     ; Done yet?
    bne.s sbRowLoop        ; Loop over rows
;                             ; Drop in to sbSolved.

sbSolved
    moveq #0,d0              ; We solved the board

sbReturn
    movem.l (a7)+,d4-d7     ; Restore callers values
    rts                     ; Done this recurse anyway!

```

Listing 5.6: Solving the Puzzle Grid

If we have run out of numbers to try, then we drop into `sbUnsolved` where we give `D0` a value of 1, which clears the Z flag, indicating a failure to solve the board, then we return to our caller via the register restoration code at `sbReturn`.

`SbNextColumn` and `sbNextRow` simply increment the column and row counters and skip back to the start of their respective loops to allow the whole grid to be scanned for missing values and attempts made to fill them.

If we manage to exit from the last row iteration, then the grid has been solved. `D0` is set to zero, which sets the Z flag to indicate a solution has been found, and we drop into `sbReturn` to return to the caller.

5.3.6 Is Digit in this Row?

The code in Listing 5.7 is used to determine if the digit in `D7` is legally able to be placed in the row in `D5`. If so, we return with the Z flag set to indicate that the digit is already present in the row. If Z is clear, then this is a valid placement for the digit in the row. It might still be found in the column or the box, but the row is fine.

```
inRow
```

```

movem.l d5/a3,-(a7)      ; Preserve board
move.w d5,d0            ; Copy row number
lsl.w #3,d5             ; Multiply by 8
add.w d0,d5             ; D0.W = Row * 9
adda.l d5,a3            ; A3 = First cell in row
moveq # gridSize-1,d0   ; 9 cells to check

irCheckCell
cmp.b (a3)+,d7          ; Number in cell already?
dbeq d0,irCheckCell     ; Keep going

;-----
; If found, D0.W = ??? and Z is set.
; If not found, D0.W = -1 and Z is clear
;-----
irDone
movem.l (a7)+,d5/a3      ; Restore board
rts

```

Listing 5.7: Checking if a digit is in a row

The code saves the working registers – all registers are preserved except for D0 – and simply multiplies the row by 9 to get the offset to the start of the row in D5 which is then added to A3 to get the offset into the puzzle data. Yes, I *could* have used MULU, but I chose not to!

D0 is then used as a counter of the number of columns to be checked, and the loop executes 9 times. If the digit in D7 is found in the row, we will drop out of the bottom of the DBEQ loop (decrement and branch *unless*³ equal) with the Z flag set.

If we exit the loop with Z clear, we ran out of columns to check and the digit is not present in the row. The working registers are restored before we return to the caller.

5.3.7 Is Digit in this Column?

The code in Listing 5.8 is used to determine if the digit in D7 is legally able to be placed in the column in D6. If so, we return with the Z flag set to indicate that the digit is already present in the row. If Z is clear, then this is a valid placement for the digit in the column. It might still be found in the box, but the row and column, if we get this far, are fine.

```

inColumn
movem.l d6/a3,-(a7)      ; Preserve board
adda.w d6,a3            ; Row 0, column D6.W
moveq #8,d0             ; 9 cells to check

icCheckCell
cmp.b (a3),d7           ; Number in cell already?
beq.s icDone            ; Yes, skip rest of loop
adda.l # gridSize,a3    ; Same cell, next row
dbeq d0,icCheckCell     ; Keep going

;-----
; If found, D0.W = ??? and Z is set.
; If not found, D0.W = -1 and Z is clear
;-----

```

³Or, if you prefer, decrement and branch *until* equal


```
icDone
    movem.l (a7)+,d6/a3          ; Restore board
    rts
```

Listing 5.8: Checking if a digit is in a column

The code saves the working registers – all registers are preserved except for D0 – and simply adds the column number in D6 to the start of the puzzle data in A3, to get the offset to the first cell of the column number in D6.

D0 is then used as a counter of the number of rows in this column to be checked, and the loop executes 9 times. If the digit in D7 is found in the column, we will drop out of the bottom of the DBEQ loop with the Z flag set.

If we exit the loop with Z clear, we ran out of rows to check and the digit is not present in the column. The working registers are restored before we return to the caller.

5.3.8 Is Digit in this Box?

This one is slightly trickier than the previous two checks. We need to check if the digit in D7 is present in the small, 3 by 3 box, containing row D5 and column D6. Listing 5.9 shows how it is done.

The code begins by saving the working registers as before. We preserve all registers except D0 in this code and return with the Z flag set to indicate that the number is in the box, or clear if not. D0 is cleared as we need to work with all 32 bits for the DIVU, and then the row number is copied over. The remainder of dividing the row number by 3 is then subtracted from the row number to get the top row number in this box. So we have $TopRow = Row - (Row \text{ Mod } 3)$.

D5, the row number, is again copied to D0 and multiplied by 9 to get the offset to add to A3 for the start of the puzzle data for the row in D5.

```
inBox
    movem.l d5-d6/a3, -(a7)
    moveq #0, d0          ; Needs to be long
    move.w d5, d0        ; D0.L = row number
    divu #3, d0          ; Row / 3
    swap d0              ; Remainder
    sub.w d0, d5         ; Row - row % 3 = box top row

    move.w d5, d0
    lsl.w #3, d0
    add.w d5, d0
    adda.l d0, a3        ; A3 = row offset of box top

    move.w d6, d0        ; D0.L = column number
    divu #3, d0          ; Column / 3
    swap d0              ; Remainder
    sub.w d0, d6         ; Box top column

    add d6, a3           ; A3 = column offset of box top

    moveq #2, d6        ; 3 rows to check

ibRowLoop
    moveq #2, d5        ; 3 columns to check
```

```

ibColLoop
  cmp.b (a3)+,d7          ; Is cell = number?
  beq.s ibDone           ; Yes, bale out
  dbra d5,ibColLoop      ; Try again

  adda.l #6,a3           ; Offset to next row in box
  dbra d6,ibRowLoop     ; Try next column

ibDone
  movem.l (a7)+,d5-d6/a3
  rts

```

Listing 5.9: Checking if a digit is in a box

In a similar vein, we need to get the starting column for the column number in D6. The same division is applied and the remainder subtracted from the column number to get the starting column for the box. Again, we have $TopColumn = Column - (Column \text{ Mod } 3)$.

This value is added to A3 to get the correct offset to the starting cell in the 3 by 3 box. D5 and D6 are now redundant, but are used to count rows and columns – and for some reason, I appear to have mixed them up, but it makes no difference, my apologies for any confusion caused — and a double DBRA loop entered to check each of the 9 cells in the box. As soon as the digit is found, we skip to `ibDone` to restore the working registers and return with the Z flag set. If we exit both loops with no sign of the digit, we return with the Z flag cleared.

5.3.9 Subroutines

All the code so far described constitutes the meat of the code to solve a puzzle. There are, a number of helpful subroutines that are called from one or more places, and those are described in this section.

Print a Byte

Listing 5.10 is a small routine to print the byte in D1 to the channel in A0. All registers are preserved and the Z flag is set if all went well, or cleared if errors were detected.

```

printByte
  movem.l d3/a1,-(a7)
  moveq #IOB.SBYT,d0
  moveq #timeout,d3
  trap #3
  movem.l (a7)+,d3/a1
  tst.l d0
  rts

```

Listing 5.10: Subroutine: Print a byte

After preserving the working registers, the `IOB.SBYT` trap call is used to send `D1.B` to channel `A0`. This requires a timeout in register `D3.W` and we use `-1` as the timeout to indicate that the trap call can take as long as it wants.

Position the Cursor

In order to preserve the heading for the console channel, where the job name is printed, we set the cursor to row 2 column 0, and the clear from there to the end of the console. Positioning the cursor

is done using the subroutine at label `at`. Listing 5.11 shows the code.

```

at
    movem.l d1-d3/a1,-(a7)
    moveq #IOW.SCUR,d0
    clr.w d1
    moveq #2,d2
    moveq #timeout,d3
    trap #3
    movem.l (a7)+,d1-d3/a1
    tst.l d0
    rts

```

Listing 5.11: Subroutine: Position the cursor at 2,0

All registers except D0 are preserved so the code begins by stacking the ones it must keep safe. The `IOW.SCUR` trap call is then set up with D1 indicating the row column number and D2 the row number. D3, as before, is the timeout for the trap.

On error, D0 will hold the error code and the Z flag will be cleared. If all went well, D0 will be zero and the Z flag will be set.

Printing Strings

To print a string, we use the code in Listing 5.12. As before, everything except D0 is preserved so we begin by saving the registers that will be corrupted.

```

printString
    movem.l d1-d3/a1,-(a7)
    move.w UT.WTEXT,a2           ; Print a string of bytes
    jsr (a2)                     ; Print it
    movem.l (a7)+,d1-d3/a1      ; Z unaffected
    rts

```

Listing 5.12: Subroutine: Printing strings

The `UT.WTEXT` vectored routine is set up next, and this expects the channel id to be in A0 and a pointer to the string to be printed in A1. The vector is loaded into A2 and executed. After restoring the working registers, we return with the Z flag already set to indicate success or failure. D0 will contain any error codes.

Opening the Console

Listing 5.13 is the console definition block, and the code – at `openScreen` – to open the console, set the ink, paper and strip colours, and apply any desired border settings.

```

conDefine
    dc.b BORDCOLOUR           ; Border colour
    dc.b BORDWIDTH           ; And width
    dc.b PAPER                ; Paper/strip colour
    dc.b INK                  ; Ink colour
    dc.w CON_W                ; Width
    dc.w CON_H                ; Height
    dc.w CON_X                ; X pos
    dc.w CON_Y                ; Y pos

openScreen

```

```

lea conDefine , a1          ; Parameters
move.w OPW.CON, a2        ; Con_ channel required
jsr (a2)                  ; Open Console & set colours
rts

```

Listing 5.13: Subroutine: Console definition and open code

The console definition block is set up using some constants defined way back at the start of the code, as shown previously in Listing 5.1. You might wish to amend the CON_X and CON_Y settings in Listing 5.1 to move the screen over a bit, if necessary on your system.

When this code is called, the application is in its starting state, so we don't really care about the registers used or corrupted. The console is opened and attributes applied using the OPW.CON vectored utility which expects A1 to be pointing at the definition block.

On exit from openScreen, the Z flag will be set if all was well, otherwise it will be clear and an error code will be in D0.

The Menu

Well, I say “menu” but it's not much of a menu – there's no vegetarian or vegan options for starters⁴. Listing 5.14 shows the code, which doesn't care about preserving registers other than A0 which holds the console channel id. Everything else is deemed expendable.

The menu is displayed starting at line 2, column 0 and offers the three options to run the demo; load a puzzle; or quit. After displaying the options, the code reads the keyboard waiting for some input. A1 is returned from waitForInput pointing at the start of the buffer, where the length of the text is stored. We need the first character in the buffer so the byte at 2(A1) is loaded into D2 and bit 5 set to convert to lower case.

D1.W holds the size of the text that was entered by the user, so if that was 1, we know the user pressed ENTER only and we are done – the demo puzzle is already loaded and nothing more needs to be done.

```

menu
    bsr.s at                ; At 2,0
    bsr.s clsScrBottom     ; Clear screen bottom
    lea messMenu, a1       ; Menu options
    bsr.s printString      ; Display menu
    bsr waitForInput       ; Get option
    move.b 2(a1), d2       ; Grab the first character
    bset #5, d2            ; Lowercase it

mnuDemo
    cmpi.w #1, d1          ; ENTER?
    beq.s mnuEnd           ; Yes, run demo

mnuQuit
    cmpi.b #'q', d2       ; Quit?
    beq die                ; Bye bye

mnuLoad
    cmpi.b #'l', d2       ; Load?
    bne.s menu             ; Invalid option
    bsr loadGame           ; Load a Sudoku to solve

```

⁴Or indeed, for main course either!

```

    beq .s mnuEnd          ; Load was ok
    lea messBadLoad , a1  ; Load failed
    bsr .s printString    ; Tell the user

mnuEnd
    bsr .s at             ; At 2,0 again
    bsr .s clsScrBottom  ; CLS
    rts                  ; Finished

```

Listing 5.14: Subroutine: Menu

Given that we now know some text was entered, we can check if D2.B is 'q' and if so, exit the application via the code at label die, which terminates the application.

If the character in D2.B is not an 'l' (lower case L) then we redisplay the menu. If it was an 'l' then we call out to loadGame to process loading a different puzzle to solve. If Z was set on return, all was well and we are done here, otherwise, we indicate that there was a loading problem and exit. In this case, the inbuilt demo will be solved.

Just prior to returning from this subroutine, the screen is once more cleared from 2,0 across and downwards – only the job name remains untouched!

Clearing Bits of the Screen

As mentioned a few times already, we want to keep the job name printed at the top of the screen. To do this we have to use two trap calls. The code is shown in Listing 5.15 and preserves A0 and D3. D0, D1 and A1 are corrupted.

```

clsScrBottom
    moveq #IOW.CLRL, d0    ; CLS cursor line
    bsr .s clsDoClear     ; Do it
    moveq #IOW.CLRB, d0   ; CLS window bottom

clsDoClear
    moveq #timeout, d3    ; Infinite timeout
    trap #3               ; Do it
    rts                   ; Ignore errors

```

Listing 5.15: Subroutine: Clearing bits of the screen

The first trap call is IOW.CLRL and will clear the screen from the cursor position to the end of the line. After this call, IOW.CLRB is used to clear all of the screen below the cursor position.

Loading Games

Puzzle files are assumed to contain ASCII representations of a grid and should contain at least 81 bytes representing the grid. This can be split as your wish desires, into separate more manageable lines, using linefeeds as and when required – those are always ignored. Any character which is not a linefeed or an ASCII digit, is considered a space for a cell to be filled in.

Loading a puzzle preserves all registers except D0. The Z flag will be set if the puzzle loaded and cleared otherwise.

Listing 5.16 shows the code executed when the user chooses to load a game from the menu. It begins by setting the cursor position and clearing the screen before prompting the user for a file name, then waits for the user to enter one. If the user simply presses ENTER, we loop around and prompt again.

```

loadGame
    bsr at ; At 2,0
    bsr.s clsScrBottom ; Clear cursorline and bottom
    lea messLoad, a1 ; Filename prompt
    bsr printString ; Prompt user
    bsr.s waitForInput ; Get a filename
    cmpi.w #1, d1 ; ENTER only?
    beq.s loadGame ; Yes, try again

```

Listing 5.16: Subroutine: Loading a puzzle - prompting the user

Once we have a potential filename, we drop into the code in Listing 5.17 where we attempt to open the file.

```

lgOpenFile
    moveq #IOA.OPEN, d0 ; Open file
    moveq #me, d1 ; This job owns it
    moveq #1, d3 ; Must exist, OPEN_IN
    move.l a0, a4 ; Save console id
    move.l a1, a0 ; Pointer to filename
    trap #2 ; Open file
    tst.l d0 ; Ok?
    beq.s lgReadFile ; Yes, read the file.
    move.l a4, a0 ; Get the console ID
    lea messOops, a1 ; File open error
    bsr printString ; Print it
    bra.s loadGame ; No, try again

```

Listing 5.17: Subroutine: Loading a puzzle - opening the file

We use the `IOA.OPEN` trap call which requires an owning job id in `D1` – I’m using `-1` for the current job – an open mode in `D3`, here I’m using the value `1` for `OPEN_IN` so the file must exist. Unfortunately, `A0` is required to point at the string containing the filename, so I have to save the console id in `A4`, which is not used or corrupted by the file opening code. It’s quicker to use a register than to stack one! `A1` currently points at the filename, so that’s copied to `A0` and the trap executed. If the file opened, we skip to `lgReadFile` to read the puzzle data, otherwise we display a message and start again by heading back to Listing 5.16.

Listing 5.18 is next, and here we set up the registers that will not change in the data reading loop. `D3` is the timeout and as usual, I’m going infinite! `A3` is the puzzle data start address for cell 0,0 and `D4` holds the count of bytes we want to read from the file, there should be 81 bytes in a puzzle. As with all `DBcc` loops, we load one less than we need.

```

lgReadFile
    moveq #timeout, d3 ; Guess!
    lea board, a3 ; Where the board is
    moveq #81-1, d4 ; 81 bytes to load

```

Listing 5.18: Subroutine: Loading a puzzle - setting up the read

At label `lgReadLoop` in Listing 5.19, we see the start of the loop to read the 81 bytes of puzzle data from the file. We use the `IOB.FBYT` trap call to read a single byte at a time into register `D1`. If we hit any errors, including EOF, before the load is complete, the program aborts – perhaps not the best idea in the world? Who fancies adding better error handling then? More homework?

If the character in `D1` is a linefeed, we ignore it and return to read the next character without affecting the counter of bytes still to be read and processed in `D4`.

```

lgReadLoop
    moveq #IOB.FBYT,d0      ; Fetch one byte
    trap #3                 ; Do it
    tst.l d0                ; Ok?
    bne die                 ; No, Give up, bale out
    cmpi.b #linefeed,d1    ; Linefeeds are ignored
    beq.s lgReadLoop       ; Read again, count untouched

lgDigits
    cmpi.b #'0',d1         ; Digit?
    bcs.s lgNonDigit       ; No, treat as space
    cmpi.b #'9',d1         ; Digit?
    bhi.s lgNonDigit       ; Still no, treat as space
    subi.b #'0',d1         ; Convert from ASCII
    bra.s lgStoreByte      ; Store in board

lgNonDigit
    moveq #0,d1            ; Non digits are zeros

lgStoreByte
    move.b d1,(a3)+         ; Save in board
    dbra d4,lgReadLoop     ; Do the rest

```

Listing 5.19: Subroutine: Loading a puzzle - reading data

At `lgDigits`, we test the character to see if it's an ASCII digit, and if so, we subtract the ASCII code for a '0' from D1. If it's not a digit, we just set D1 to zero at `lgNonDigit` and in both cases, end up at `lgStoreByte` where we write the byte in D1 into the puzzle grid at A3, incrementing A3 to point at the next cell's data space.

After storing the byte, the counter is decremented and if there is more to do, we skip back to read another byte. If we have read all 81 bytes from the file, then we close it using `IOA.CLOS` at label `lgCloseFile`, in Listing 5.20.

```

lgCloseFile
    moveq #IOA.CLOS,d0      ; Close file
    trap #2                 ; Do it, no errors in SMSQ/E
    moveq #0,d0             ; No errors – QDOS either
    move.l a4,a0            ; Restore console ID
    rts

```

Listing 5.20: Subroutine: Loading a puzzle - closing the data file

In QDOS, closing a file which was not open results in a “File not open” error, but in SMSQ/E, there is no error reported. The code in Listing 5.20 assumes the latter case and clears D0 to indicate no errors.

The console is copied back from A4 to A0 before we return to the calling code, as we will be writing to the console soon and need it!

It should be noted that loading a game makes no attempt to determine if the data are valid. You could load a puzzle with repeating characters in rows, columns and/or boxes, it doesn't care. What happens in this case? The code sill still try to solve the puzzle, but will run for quite some time, and will have to be aborted, probably, when you get fed up!

Reading Input From the Keyboard

Listing 5.21 shows the code used to obtain input from the user. Any time we call this subroutine, the user is expected to press ENTER. This character will be saved as a linefeed in the input buffer and the count of characters obtained, in D1, will include the linefeed.

```

waitForInput
    moveq #IOB.FLIN, d0          ; Fetch input with Linefeed
    move.w #BUFFERSIZE, d2      ; How big is my buffer?
    lea inputBuffer, a1         ; Input buffer space
    move.l a1, a4               ; Save buffer
    addq.l #2, a1               ; Skip where the length word is
    trap #3                     ; Fetch input D1.W = size
    tst.l d0                    ; Any errors?
    bne.s waitForInput         ; Yes, just try again
    move.w d1, (a4)             ; Save the string length
    subq.w #1, (a4)             ; Lose the linefeed
    move.l a4, a1               ; Buffer start
    rts

inputBuffer ds.b BUFFERSIZE+2  ; Input buffer
            ds.w 0              ; Alignment

```

Listing 5.21: Subroutine: Getting user input

We use the IOB.FLIN trap call to fetch a line of input. D2 tells the trap how big the buffer is and sets the maximum number of bytes that can be returned. A1 points at the buffer start and this is copied to A4 as we need to store the length word there. A1 is incremented by 2 to point at the position where we want the data to be read into. D3 already holds the timeout.

After the trap call, if there were errors, we skip back and attempt to get input from the user again. If no errors were detected, the length word is stored in the start of the buffer but then decremented to delete the trailing linefeed character, which we know is definitely there. A4 is copied to A1 to return the start of the buffer to the caller.

If the user types more characters than will fit in the buffer, then an overflow error will be returned and the final character in the buffer will not be the linefeed. We don't specifically trap this error in the code above as we simply reattempt in the case of errors being detected.

The code expects A0 to be the console id and trashes just about everything else. A1 is returned pointing to the start of the buffer.

Scanning for ESC

I mentioned that puzzle data are not validated when loaded. If a bad data set is loaded, then the code could run for some time and we need a manner of aborting a run. The code in Listings 5.22 and 5.23 carry out a quick scan of the keyboard, looking for the ESC key.

Please note, this is the same as calling KEYROW in S*BASIC and it's possible that pressing ESC in another application will be detected by the Sudoku solver, and abort its current run. Note also, Listing 5.23 is out of line here, it's at the end of the file in reality, but is listed here as it pertains to this subroutine.

```

scanForESC
    movem.l d0-d1/d5/d7/a3, -(a7) ; Save corrupted registers
    moveq #SMS.HDOP, d0           ; Do hardware op
    lea ipcCommand, a3           ; Command to execute

```



```

trap #1 ; Scan keyboard , no errors
btst #escKey , d1 ; ESC bit will be 0 now
movem.l (a7)+, d0-d1/d5/d7/a3 ; Restore corrupted registers
rts

```

Listing 5.22: Subroutine: Scanning for ESC

Listing 5.22 saves all registers. Only the Z flag is affected and is set if ESC was not pressed. We use the SMS.HDOP trap call to run a hardware operation to scan the keyboard for ESC, but we can also detect ENTER, SPACE, '\ ' and the 4 arrow keys with the parameters in Listing 5.23 which are pointed at by register A3. After the trap, bit 3, the ESC key bit, is tested and if set, ESC was pressed. If ESC was not pressed, the bit will be clear and as such, the Z flag will be set.

```

ipcCommand
dc . b 9 , 1 , 0 , 0 , 0 , 0 , 1 , 2

```

Listing 5.23: Subroutine: KEYROW command for ESC

Listing 5.23 is the parameter set required by SMS.HDOP to scan KEYROW(1) on the keyboard.

The Demo Board

If the user simply presses ENTER at the menu, then the inbuilt demo puzzle will be solved. Listing 5.24 shows the puzzle's data bytes where a zero is used to indicate an empty cell. Note that the data are *not* ASCII, they are binary values.

```

board
dc . b 9 , 0 , 0 , 5 , 3 , 1 , 0 , 0 , 0 ; Index 0 to 8
dc . b 0 , 0 , 0 , 0 , 4 , 9 , 3 , 8 , 0 ; Index 9 to 17
dc . b 0 , 0 , 1 , 8 , 0 , 7 , 4 , 0 , 0 ; Index 18 to 26 , H-Sep here
dc . b 7 , 0 , 4 , 2 , 0 , 0 , 0 , 0 , 6 ; Index 27 to 35
dc . b 0 , 0 , 3 , 0 , 9 , 0 , 0 , 0 , 7 ; Index 36 to 44
dc . b 0 , 6 , 9 , 0 , 0 , 3 , 0 , 0 , 0 ; Index 45 to 53 , H-Sep here
dc . b 0 , 0 , 7 , 0 , 0 , 0 , 0 , 0 , 0 ; Index 54 to 62
dc . b 0 , 9 , 2 , 0 , 0 , 0 , 0 , 7 , 0 ; Index 63 to 71
dc . b 5 , 1 , 0 , 0 , 0 , 4 , 0 , 0 , 8 ; Index 72 to 80

ds . w 0

```

Listing 5.24: Subroutine: Demo puzzle data

There is an alignment check after the puzzle's data as 81 bytes would leave a dangling odd address. This is not a problem on 68020 emulators, but will cause no end of problems for the old bare bones QL, so is best avoided.

The puzzle data is shown here as separate rows of 9 bytes each, but there are no linefeeds in the data, so it is simply a long string of 81 bytes of data. Don't let the way it's printed confuse how it is actually stored.

This 81 byte area is where the main code looks when solving a puzzle. If the user loads a puzzle, it will be loaded here too, overwriting the demo data.

Messages

Various messages used throughout the application are shown in Listings 5.25 and 5.26. Listing 5.25 is the horizontal separator used by the printBoard subroutine.

```

hSeparator
    dc.w hSepEnd--2
    dc.b '---+---+---',linefeed
hSepEnd equ *
    ds.w 0

```

Listing 5.25: Subroutine: Grid horizontal separator

Listing 5.26 contains all the other messages.

```

messMenu
    dc.w mMenuEnd--2
    dc.b 'ENTER: Load demo',linefeed
    dc.b 'L: Load game',linefeed
    dc.b 'Q: Quit.',linefeed,linefeed
    dc.b 'Please choose:'
mMenuEnd equ *
    ds.w 0

messBadLoad
    dc.w mBadLoadEnd--2
    dc.b 'Failed to load puzzle.'
mBadLoadEnd equ *
    ds.w 0

messSuccess
    dc.w mSucEnd--2
    dc.b linefeed,'Solved.',linefeed,linefeed
mSucEnd equ *
    ds.w 0

messFailed
    dc.w mFailEnd--2
    dc.b linefeed,'Unsolvable!!',linefeed
mFailEnd equ *
    ds.w 0

messLoad
    dc.w mLoadEnd--2
    dc.b 'File name',linefeed
mLoadEnd

messEnter
    dc.w mEnterEnd--2
    dc.b 'Press ENTER to quit:'
mEnterEnd equ *
    ds.w 0

messOops
    dc.w messOopsEnd--2
    dc.b 'Oops. File open failed.'

```

```
messOopsEnd equ *  
ds .w 0
```

Listing 5.26: Subroutine: Messages



6. 32 Bit Multiplication

I shall confess straight away, I am not a mathematician, nor do I play one on TV. There may therefore be some weirdness in this chapter. Feel free to correct or update me!

I don't know about you, but when I see code to do multiplication (or worse, division) in Assembly Language, I tend to gloss over and skip the meaty stuff. For some reason, I get a mental block. So I decided to do something about it and find out if I could make myself understand it.

Obviously, I'm starting simple with an *unsigned* 32 bit by 32 bit multiplication routine which takes two registers as the inputs, D0 and D1, and returns the product in D3 and D2 with D3 being the high 32 bits.

6.1 The MC68008 and MC68020

I am aware that the MC68020 already has a 32 bit by 32 bit multiply instruction, before anyone starts writing an email! The MC68008 only has a 16 bit by 16 bit multiply instruction, and this code is, after all, a learning experience for me. And if I have to learn it, then don't think you are getting off so easily!

6.2 The Theory

In decimal, if we want to multiply the values 24 and 32, we don't usually do the multiplication in one calculation of $24 * 32$ – we split it up into manageable chunks and perform four multiplications and one addition, of the four results, to get the final product:

$$\begin{aligned}
2 * 4 &= 8 \\
2 * 2 * 10 &= 40 \\
3 * 10 * 4 &= 120 \\
3 * 10 * 2 * 10 &= 600 \\
8 + 40 + 120 + 600 &= 768
\end{aligned}$$

In general, we end up with the formula:

$$(U_1 * U_2) + (U_2 * T_1 * Base) + (T_2 * Base * U_1) + (T_1 * Base * T_2 * Base)$$

which can be rearranged into the following:

$$(U_1 * U_2) + (U_2 * T_1 * Base) + (T_2 * U_1 * Base) + (T_1 * T_2 * Base^2)$$

Where:

- ‘ T_1 ’ and ‘ T_2 ’ represent the “tens” digits of the two numbers;
- ‘ U_1 ’ and ‘ U_2 ’ represent the “units” digits of the two numbers;
- ‘Base’ is the base of the number system in use, 10 in this example.

Substituting our two values, 24 and 32, into the equation, we get:

$$(2 * 4) + (2 * 2 * 10) + (3 * 4 * 10) + (3 * 2 * 100) = 768$$

So far so good! It seems to work perfectly. Can we do the same thing with our 32 bit register values? Yes we can! If you consider that a 32 bit register value is really just $D_{high} * 65,536 + D_{low}$ then we can surely substitute these values into our equation. The number base is therefore 65,536 in this case. As multiplying by 65,536 is simply a 16 bit shift left, then we end up with the following equation:

$$(D0_{lo} * D1_{lo}) + (D0_{lo} * D1_{hi} \ll 16) + (D0_{hi} * D1_{lo} \ll 16) + (D0_{hi} * D1_{hi} \ll 32)$$

Of course, we don’t have to use the 16 bit words as our “digits”, we could easily¹ use the 4 bytes in each register as our “digits”, but given that the MC68008 has a 16 bit multiply, then we might as well use it.

6.3 The Code

Listings 6.1 through 6.10 show the code necessary to convert the above theory into actual values.

¹For certain values of “easily”!

```

mult32
  movem.l d4-d5, -(a7)      ; Save Working registers
  moveq #0, d3              ; Assume result will be zero
  move.l d3, d2

```

Listing 6.1: Unsigned multiplication - initialisation

Listing 6.1 is the initialisation code for the unsigned multiplication routine. It saves the working registers on the stack, and initialises the result, in D3:D2, to zero.

```

mult32TestD0D1
  tst.l d0                  ; Is D0 zero?
  beq.s m32Exit             ; Product must be zero, done
  tst.l d1                  ; Is D1 zero?
  beq.s m32Exit             ; Product must be zero, done

```

Listing 6.2: Unsigned multiplication - testing for zero

The code continues in Listing 6.2 where both input registers are tested to ensure that neither is holding the value zero. If one of the registers is found to be zero, then the code makes a rapid exit as the result will also be zero and this has been initialised already.

```

m32Step_1
  move.w d0, d2             ; D2 = D0 low
  mulu.w d1, d2            ; D2 = (D0 low * D1 low)

;-----
; We also need D0 low in D4.
;-----
  move.w d0, d4            ; D4 = copy of D0 low

```

Listing 6.3: Unsigned multiplication - step 1

As we have two non-zero values, we have to do the multiplication. We begin with step one of four, and multiply the two low words of registers D0 and D1 together. Listing 6.3 shows the code and you will note that the product of the two low words is saved in register D2.L, which is the low 32 bits of the 64 bit result.

As we need the low word of D0 again shortly, it is copied over to register D4.

```

m32Step_4
  swap    d0                ; D0 = LLLLHHHH
  swap    d1                ; D1 = LLLLHHHH
  move.w  d0, d3            ; D3 = D0 high
  mulu.w  d1, d3            ; D3 = (D0 high * D1 high)

```

Listing 6.4: Unsigned multiplication - step 4

Listing 6.4, which may appear out of sync, is the fourth of four steps. Don't worry, steps two and three will be here soon! In this code, we are multiplying the two high words of the input values together. To do this, we have to swap the top 16 bits to the bottom as the MULU instruction works on the lowest 16 bits of the input values.

The result is saved in register D3.L which is the high 32 bits of the 64 bit result.

You may be wondering about the step two and three. They are pretty much identical. However, there's a catch. We need to calculate the product of the high word of one register and the low word of the other, which gives a 32 bit result. That's easy enough, but the catch is this; we have to shift this result 16 bits leftwards before adding it to D3:D2.

We only have 32 bits in a register, and we need 48!

Listing 6.5 shows the code for step two.

```
m32Step_2
    mulu .w  d1 , d4          ; D4 = (D0 low * D1 high)
    swap    d4                ; D4 = LLLLHHHH
    moveq   #0 , d5          ; D5 = 00000000
    move .w  d4 , d5          ; D5 = (D0lo * D1hi) 0000HHHH
    clr .w   d4              ; D5:D4 = 0000HHHH LLLL0000
```

Listing 6.5: Unsigned multiplication - step 2

When we reach this point, D4.W is holding the low word of the value passed in register D0.L, and D1 has had it's two words swapped around so that the current low word was originally the high word of the passed in value. We multiply these two together and save the result in D4.L.

We now need to shift this value 16 bits leftwards, and to do this, we employ register D5. D4 is swapped to put the high word into the low word, and this is copied to D5 prior to clearing the low word of D4. This leaves the register pair D5:D4 holding the value \$0000HHHH:LLLL0000 where HHHH and LLLL are the high and low words of the product of D0 low and D1 high words. (I wish I could write that in better English!)

The intermediate result in D5:D4 needs to be added to D3:D2 holding the result of the multiplication so far. Listing 6.6 is the code to do this addition.

```
m32Step_2add
    add .l   d4 , d2
    addx .l  d5 , d3          ; D3:D2 = Steps 1, 2 and 4 now
```

Listing 6.6: Unsigned multiplication - step 2 addition

Adding the two low registers first might generate a carry, which will also set the X (Extended) flag. This is added in to the addition of the two high registers to ensure that the carry is properly propagated into the high 32 bits of the result so far.

Step three of the calculation comes next and it is very similar to step two, which we have just seen. This time the calculation is D0 high multiplied by D1 low and again, shifted 16 bits leftwards using registers D5:D4 to hold the intermediate result.

```
m32Step_3
    swap    d1                ; D1 = HHHHLLLL again
    move .w  d0 , d4          ; D4 = D0 high word, ????HHHH
    mulu .w  d1 , d4          ; D4 = (D0hi * D1lo) HHHHLLLL
    swap    d4                ; D4 = LLLLHHHH
    moveq   #0 , d5          ; D5 = 00000000
    move .w  d4 , d5          ; D5 = (D0hi * D1lo) 0000HHHH
    clr .w   d4              ; D5:D4 = 0000HHHH LLLL0000
```

Listing 6.7: Unsigned multiplication - step 3

```
m32Step_3add
    add .l   d4 , d2
    addx .l  d5 , d3          ; We have a result!
```

Listing 6.8: Unsigned multiplication - step 3 addition

Listings 6.7 and 6.8 show the code to carry out the multiplication and addition. There's not much difference between Listing 6.7 and Listing 6.5 other than swapping the words in D1 around to get them where we need them.

As we want the two input registers D0 and D1 to be preserved, we need to do a quick swap of register D0 to accommodate this. Listing 6.9 shows the one liner that does the needful.

```
m32Done
    swap d0                ; D0 finally back to HHHHLLLL
```

Listing 6.9: Unsigned multiplication - restoring D0

```
m32Exit
    movem.l (a7)+,d4-d5    ; Restore working registers
    rts
```

Listing 6.10: Unsigned multiplication - all done

All that remains to be done is to restore the working registers and exit. Listing 6.10 shows the necessary code.

6.4 What About Signed Values?

Signed multiplication is *almost* as simple as unsigned, with a small caveat. If the two values have different signs – one negative and one positive – then the product is negative. If both have the same sign – both negative or both positive – the product is always positive.

In the unsigned code discussed in this chapter, multiplying -2 by -1, \$FFFFFFFE by \$FFFFFFF, results in D3:D2 holding \$FFFFFFFD:00000002 when it should simply be \$00000000:00000002. Also, the N flag is set to indicate a negative number. This is clearly wrong so we need a better bit of code for signed multiplication.

Listings 6.11 through 6.16 show the new code that I have added *above* the mult32 code previously discussed in Listings 6.1 through 6.10.

```
smult32
    move.w d6,-(a7)        ; Save flag register
    moveq #1,d6           ; Assume both have same sign
```

Listing 6.11: Signed multiplication - initialisation

The initialisation code, in Listing 6.11, simply saves register D6 on the stack and then initialises it. D6.B is used as a flag to determine if any adjustments for the signs of the two values being multiplied, are necessary. It is initially set to a value of +1 to indicate no adjustment required.

```
sm32TestD0
    tst.l d0                ; Check for negative
    bpl.s sm32TestD1       ; Skip if positive
    neg.b d6                ; D6 is negative if D0 is too
    neg.l d0                ; Make D0 positive
```

Listing 6.12: Signed multiplication - testing D0

```
sm32TestD1
    tst.l d1                ; Check for negative
    bpl.s sm32DoMult       ; Skip if positive
    neg.b d6                ; D6 now holds sign of result
    neg.l d1                ; Make D1 positive
```

Listing 6.13: Signed multiplication - testing D1

Listings 6.12 and 6.13 show the code which checks both input values for negativity. If one or both of the register values is negative, then the flag byte in D6 will be negated once for each negative value, and the corresponding register value made positive.

```
sm32DoMult
    bsr .s mult32                ; Do unsigned multiplication
    tst .b d6                    ; Test flag byte
    bpl .s sm32Exit              ; Result is positive , all done
```

Listing 6.14: Signed multiplication - doing the multiplication

After checking the two values, we have a guarantee of two positive values in the two registers. As shown in Listing 6.14 we call the unsigned multiplication code already discussed.

```
sm32Adjust
    neg .l d2                    ; Negate the low word
    negx .l d3                   ; And then the high word
```

Listing 6.15: Signed multiplication - adjusting the result

On return, we test D6.B and if it is positive, no adjustment of the result is necessary. If it is negative, we drop into Listing 6.15 to adjust the result, this is simply a case of negating the low register, D2, and then negating with extend (aka carry) the upper register, D3.

```
sm32Exit
    move .w (a7)+, d6            ; Restore flag register
    rts                          ; All done
```

Listing 6.16: Signed multiplication - termination

All that remains to do is to restore D6 and return to the caller with the result in registers D3:D2. Listing 6.16 has the code.

6.5 A Final Warning

So, that's multiplication in a nutshell. I hope I've managed to explain it in an understandable manner. It's certainly taken me long enough to get up the will to understand it myself! Bear in mind that when using the inbuilt multiplication instructions of the MC68020 that the flags will be correctly set, however, the code in these routines might not be as accurate. The following is what I found in testing.

- The Z (Zero) flag will be correctly set if one or more of the input values is zero, or if some weird combination of values results in a zero product. Is such a thing valid?
- The N (Negative) will be correctly set in either signed or unsigned multiplication, if the top bit of the result is a 1. That's bit 31 of register D3.
- The V (Overflow) flag will be set if the multiplication overflows. Unsigned multiplication of two 32 bit values must always result in a 64 bit product, I can't get it to overflow into 65 bits. I suspect therefore that there's a problem with signs. My suspicion is that the overflow flag will be set if a product overflows into the top bit of register D3 thus becoming negative when it should be positive.
- The C (Carry) and X (Extend) flags *should* be zero. So far in testing, they have been.

Oh yes, I mentioned that MC68020 has a 32 bit multiplication. All of the code in Listings 6.1 through 6.10 and 6.11 through 6.16 can be replaced by:

```
mult32
    move .l d0 , d3
    mulu .l d1 , d3 : d2
    rts
```

Listing 6.17: MC68020 - unsigned 32bit multiplication

and:

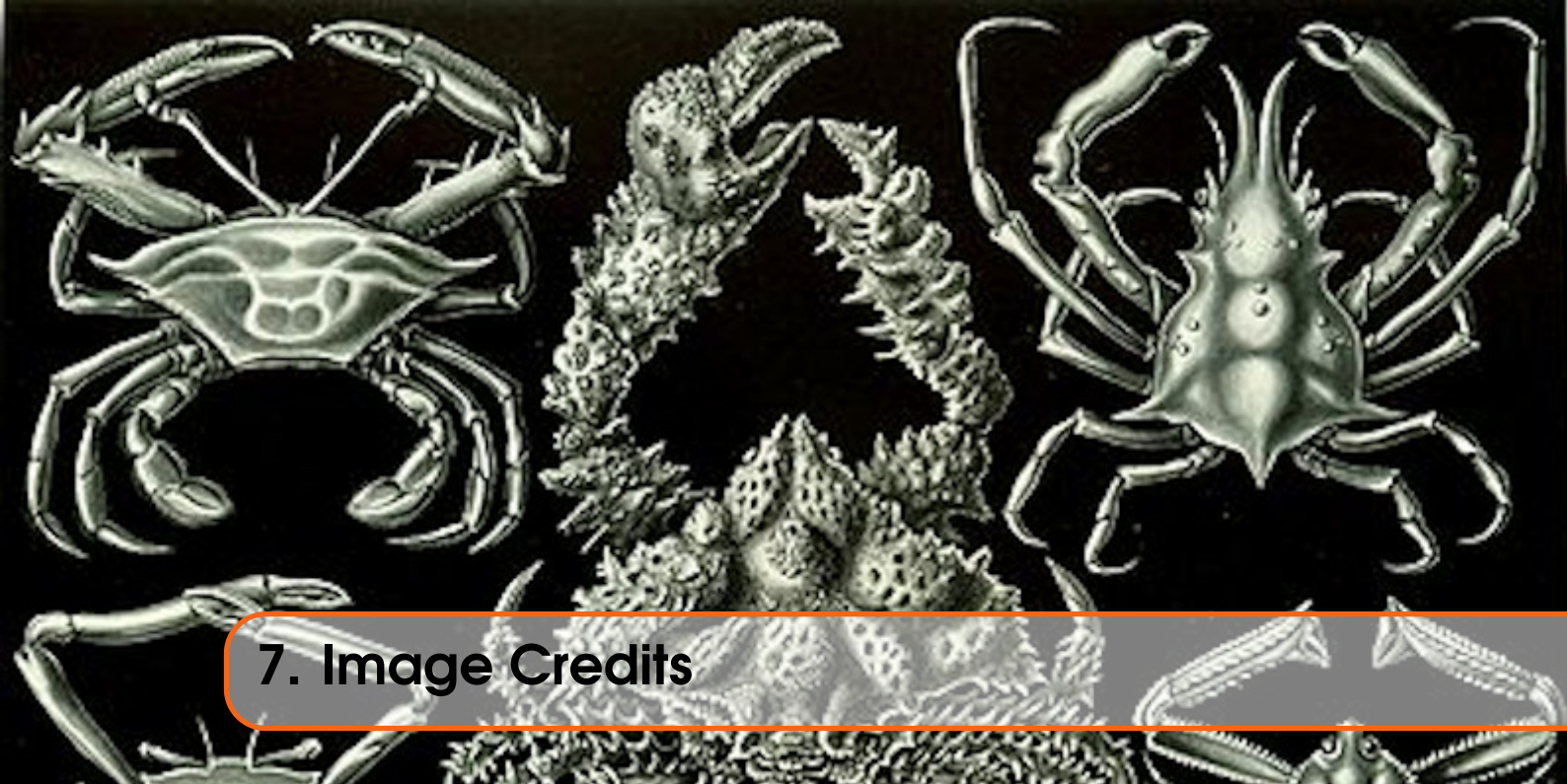
```
smult32
    move .l d0 , d3
    muls .l d1 , d3 : d2
    rts
```

Listing 6.18: MC68020 - signed 32bit multiplication

And, the flags will always be correctly set afterwards.

6.6 And Finally

No! No! No! Not under any circumstances! I flatly refuse to do the same for 32 bit division by a 32 bit number. Sorry, no can do! If anyone wants a guest chapter in a forthcoming issue, be my guest and write me a chapter on 32 bit division!



7. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Decapods*. The Decapoda or decapods (literally "ten-footed") are an order of crustaceans within the class Malacostraca, including many familiar groups, such as crabs, lobsters, crayfish, shrimp and prawns. Most decapods are scavengers. The order is estimated to contain nearly 15,000 species in around 2,700 genera, with around 3,300 fossil species.

I have also cropped the cover image for use on each chapter heading page.

You can read about Decapods on [Wikipedia](#) and there is a brief overview of the above book, also on [Wikipedia](#), which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Decapods have absolutely nothing to do with the QL or computing in general - in fact, I suspect many of them died out before electricity was invented, and the rest probably don't care about electricity or computers! However, I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are 10 legged crustaceans.